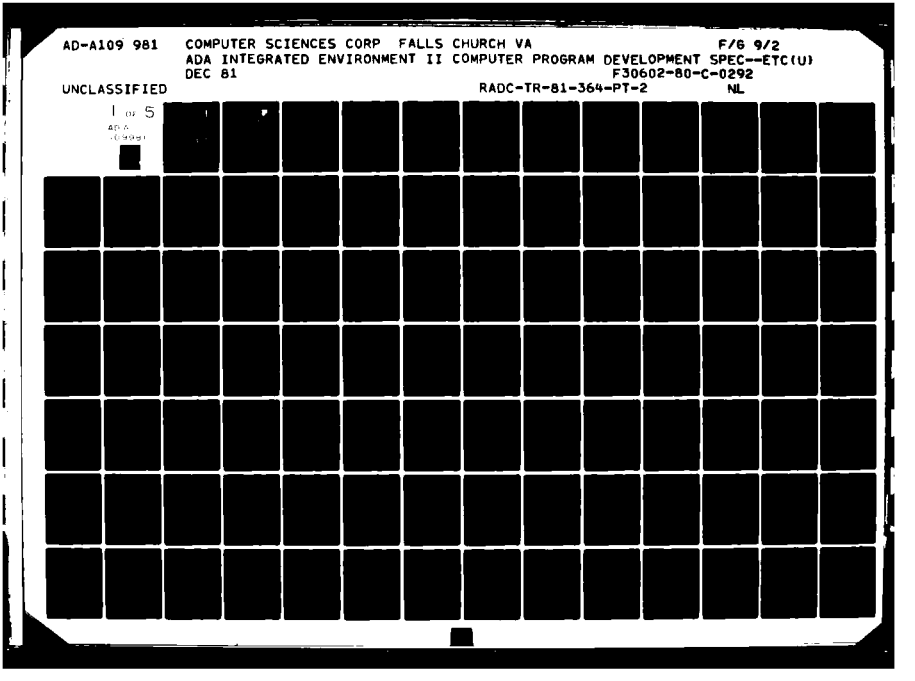
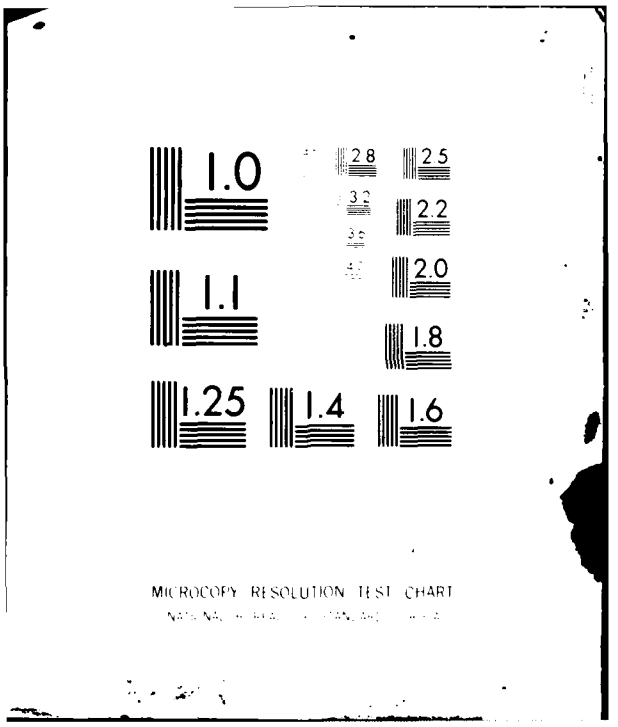
AD-A109 981 COMPUTER SCIENCES CORP FALLS CHURCH VA F/G 9/2
ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPEC--ETC(U)
DEC 81 F30602-80-C-0292
UNCLASSIFIED RADC-TR-81-364-PT-2 NL




1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

PHOTOGRAPH THIS SHEET

AD A109981

DTIC ACCESSION NUMBER

LEVEL

INVENTORY

Computer Sciences Corp
Falls Church, VA

ADA Integrated Environment II Computer Program Development Specification, Interim Rpt. 15 Sep. 80 - 15 Mar. 81

DOCUMENT IDENTIFICATION

Contract F30602-80-C-0292 RADC-TR-81-364, Part 2

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

DISTRIBUTION STATEMENT

ACCESSION FOR	
NTIS	GRA&I
DTIC	TAB
UNANNOUNCED	
JUSTIFICATION	
BY	
DISTRIBUTION /	
AVAILABILITY CODES	
DIST	AVAIL AND/OR SPECIAL
A	

DISTRIBUTION STAMP

DTIC COPY INSPECTED 3

DTIC
ELECTE
JAN 25 1982
S D
D

DATE ACCESSIONED

82 01 12 002

DATE RECEIVED IN DTIC

PHOTOGRAPH THIS SHEET AND RETURN TO DTIC-DDA-2

DTIC FORM OCT 79 70A

DOCUMENT PROCESSING SHEET

RADC-TR-81-364, Part 2
Interim Report
December 1981

ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

Computer Sciences Corporation

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-81-364, Part 2 has been reviewed and is approved for publication.

APPROVED:

DONALD F. ROBERTS
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-81-364, Part 2	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPECIFICATION		5. TYPE OF REPORT & PERIOD COVERED Interim Report 15 Sep 80 - 15 Mar 81
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s)		8. CONTRACT OR GRANT NUMBER(s) F30602-80-C-0292
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Sciences Corporation 803 Broad Street Falls Church VA 22046		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62204F/62702F/33126F 55811918
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COES) Griffiss AFB NY 13441		12. REPORT DATE December 1981 13. NUMBER OF PAGES 442
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Donald F. Roberts (COES)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada	MAPSE	AIE
Compiler	Kernel	Integrated environment
Database	Debugger	Editor
KAPSE	APSE	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The Ada Integrated Environment (AIE) consists of a set of software tools intended to support design, development and maintenance of embedded computer software. A significant portion of an AIE includes software systems and tools residing and executing on a host computer (or set of computers). This set is known as an Ada Programming Support Environment (APSE). This B-5 Specification describes, in detail, the design for a minimal APSE, called a MAPSE. The MAPSE is the foundation upon which an

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

APSE is built and will provide comprehensive support throughout the design, development and maintenance of Ada software. The MAPSE tools described in this specification include an Ada compiler, linker/loader, debugger, editor, and configuration management tools. The kernel (KAPSE) will provide the interfaces (user, host, tool), database support, and facilities for executing Ada programs (runtime support system).

INTRODUCTION

This document presents the Computer Program Development Specifications (Type B5) for the Computer Program Configuration Items (CPCIs) for the CSC/SEA design of the Ada Integrated Environment (AIE) under Rome Air Development Center (RADC) Contract Number F30602-80-C-0292. These specifications are comprised of the following volumes:

PART I:

Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.

Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.

PART II:

Volume 3, Computer Program Development Specification for CPCI APSE Command Language Interpreter.

Volume 4, Computer Program Development Specification for CPCI MAPSE Configuration Management System.

Volume 5, Computer Program Development Specification for CPCI Ada Compiler.

Volume 6, Computer Program Development Specification for CPCI MAPSE Linker.

Volume 7, Computer Program Development Specification for CPCI MAPSE Editor.

Volume 8, Computer Program Development Specification for CPCI MAPSE Debugger.

Accompanying this document is an Interim Technical Report (ITR), which describes the principles influencing the preliminary design and provides the rationale for the decisions made, and the System Specification (Type A), which presents the functional requirements for the AIE.

Table 1 provides a cross-reference between the AIE Statement of Work (SOW) and the specifications.

2

PHASE I SOW REQUIREMENTS	A - SPEC.	B5 - SPEC.
4.1 Phase I Design		
4.1.1 General Requirements	3.1.1	
4.1.1.1 Data Base Support, Interfaces to host facilities (H.W. & S.W.), user interfaces, tool interfaces	3.1.1.1 3.1.1.2 3.1.4 3.7.1	KDBS - 3.2.5 3.3 ACLI - 3.2.5 3.3
4.1.1.2 Portable to maximum extent possible, External interfaces should be clearly isolated, clearly identified	3.1.1.1 3.1.1.2 3.1.2 3.1.4 3.1.5 3.1.5.2	KFW - 3.1.1 KDBS - 3.1.1 ACLI - 3.1.1 CMS - 3.1.1 Compiler - 3.1.1 Linker - 3.1.1 Editor - 3.1.1 Debugger - 3.1.1
4.1.1.3 Specify uniform protocol conventions between user, tools and MAPSE/KAPSE, formats for invoking KAPSE/MAPSE facilities should be uniform or identical	3.1.5.1 3.1.5.2	KDBS - 3.3 KFW - 3.2.5 ACLI - Command Utilities
4.1.1.4 Shall include features to protect itself from user and system errors	3.2.3 3.2.5 3.3.7	KDBS - 3.2.5.7 3.2.5.8 3.3.6 3.3.7
4.1.1.5 Software should be modular and reusable	3.7	KFW - 3.1.1 KDBS - 3.1.1 ACLI - 3.1.1 CMS - 3.1.1 Compiler - 3.1.1 Editor - 3.1.1 Linker - 3.3.1 Debugger - 3.1.1

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.2 KAPSE DATA BASE REQUIREMENTS		
4.1.2.1 Capability to create, delete, modify, store, retrieve, input, and output data base objects	3.7.1.2	KDBS - 3.2.5.6 3.3.4
4.1.2.2 Shall provide for all forms of data necessary to fulfill all SOW Requirements	3.7.1.2	KDBS - 3.2.5.3
4.1.2.3 Shall not be dependent on external software systems not part of the host operating system	3.7.1.2	KDBS - 3.2.5
4.1.2.4 Support creation and storage of Ada libraries in source form	3.7.1.2 3.7.2	KDBS - 3.2.5 3.3.1 3.3.4
4.1.2.5 Capability to define new categories of objects without imposing restrictions on computer information stored in objects	3.7.1.2	KDBS - 3.2.5.3 3.3.1
4.1.2.6 Provide flexible storage facilities to all MAPSE tools. Capability to read and write data base objects from within any MAPSE tool using data transfer and control functions and high level I/O function	3.7.1.2	KDBS - 3.2.5.6 3.3.4
4.1.2.7 Capability to create partitions	3.7.1.2	KDBS - 3.2.5.1 3.3.2

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.2.8 Capability to assign version qualifiers to objects or groups of objects. Time/Date and serial number. Capability to designate and use default version	3.7.1.2	KDBS - 3.2.5.4 3.3.1
4.1.2.9 Capability to create object attributes: History, Category and Access.	3.7.1.2	KDBS - 3.2.5 3.2.5.1 3.2.5.2 3.2.5.4 3.2.5.3 3.2.5.5 3.3.1 3.3.3
4.1.2.10 Capability to control access to data base objects using version qualifier, attributes, and partitions. "Programmable" access controls; provision for privileged user.	3.7.1.2	KDBS - 3.2.5.1 3.2.5.2 3.2.5.3 3.2.5.4 3.2.5.5 3.2.5.6 3.3.1 3.3.3
4.1.2.11 Capability to archive data base objects	3.7.1.2	KDBS - 3.2.5.7 3.2.5.8 3.3.6 3.3.7
4.1.2.12 Data base resources and operations as a result of this effort shall be available to Ada programmers	3.7.1.2	KDBS - 3.2.5 3.3
4.1.3 KAPSE INTERFACE REQUIREMENTS		
4.1.3.1 Specifiy virtual interface for KAPSE /MAPSE communication	3.1.5.2 3.7.2.1	KDBS - 3.2.4 3.2.5 3.3 3.2.4.1

5'

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.3.2 Virtual interface will provide user capability to invoke MAPSE tools, interact and exercise control over invoked tool	3.1.5.2 3.7.2.1	KFW - 3.2.4.1 KDBS - 3.2.4
4.1.3.3 Virtual interface will have the capability to invoke any MAPSE tool from other MAPSE tool	3.1.5.2	KFW - 3.2.4.1 ACLI - 3.1.3 3.2.4.2
4.1.3.4 Virtual interface will provide the capability for user LOGON/LOGOFF INITIATE/TERMINATE functions	3.1.5.3 3.7.2.1	KFW - 3.3.2 3.2.5.2
4.1.3.5 Virtual interface will provide the capability to execute Ada programs	3.1.5.1 3.7.2.1	ACLI - 3.1.1 3.2.4.1
4.1.3.6 User commands for job control and invoking tools shall have a uniform format	3.1.5.1 3.7.2.1	ACLI - 3.2.5
4.1.3.7 User communication at command level will be possible in standard Ada character set	3.1.5.1	ACLI - 3.1.1
4.1.3.8 Provide standard terminal interface specifications and functions to facilitate batch and interactive terminals. Specification will include protocols for synchronous user interactions and standards for implementing simple editing of the command line	3.1.5.3	KFW - 3.3.1 3.3.9 3.4.1.2.8

6

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.3.9 Specify host interfaces to support low-level I/O function and high level I/O package	3.7.1.1	KFW - 3.2.5.8 3.2.5.9 3.3.5.10 3.3.9 3.3.10
4.1.3.10 Specify data identified as shared data and provide as standard interfaces	3.1.5.2 3.7.2	KDBS - 3.2.5.6 3.3.4
4.1.4 KAPSE FUNCTIONS		
4.1.4.1 Provide basic Run-time support facilities	3.1.1.1	KDBS - 3.2.5.6 3.3.4
4.1.4.2 Provide basic data transfer and control functions to support high level I/O package	3.1.1.1 3.7.1.1	KDBS - 3.2.5.6 3.3.4
4.1.5 GENERAL MAPSE REQUIREMENTS		
4.1.5.1 Tools written in Ada and conform to standard interface specifications	3.7.2	Compiler - 3.1.1 Editor - 3.1.1 Debugger - 3.1.1 CM - 3.1.1 ACLI - 3.1.1 Linker - 3.1.1
4.1.5.2 Inter-tool communication via virtual interface facilities	3.1.5.2 3.7.2	KDBS - 3.2.4 KFW - 3.2.4.1 ACLI - 3.1.3

7

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.5.3 Formats for similar user commands shall be uniform and consistent across all tools	3.1.5.1	ACLI - 3.1.1
4.1.5.4 Data produced by one MAPSE tool needed or useful to another tool shalled be saved. Identify such data and provide interface specifications	3.1.5.2 3.7.2	ACLI - Appendix ACL Compiler - Appendix A Appendix C Appendix D Linker - 3.3.2.3 Appendix C
4.1.6 MAPSE Editor, includes the following capabilities: find, alter insert, delete, input, output, move copy, and substitute	3.7.2.5	Editor - 3.2.5 3.3
4.1.7 MAPSE Debugger		Debugger
4.1.7.1 Shall function at the Ada level	3.7.2.6	Debugger - 3.2.5
4.1.7.2 Shall support debugging of all Ada language features including concurrent programs	3.7.2.6	Debugger - 3.2.5 3.3.2 3.3.15
4.1.7.3 Shall provide linkage between executing program in binary form and corresponding source program	3.7.2.6	Compiler - Appendix C Debugger - 3.2.5 3.2.6 Linker - 3.2.5.3 Editor - 3.3

8

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.7.4 As a minimum shall provide: Breakpoints Display Values Modify Values Display and modifications of variables shal be machine or scalar type representations at the users option Display Subprogram arguments Modify flow of program Tracking Dumps	3.7.2.6	Debugger - 3.2.4
4.1.8 Compiler Requirements		Compiler
4.1.8.1 Operate in a modular fashion; minimize resource utilization	3.7.2.3	Compiler - 3.3
4.1.8.2 Operate in batch, remote batch, and on-line modes	3.7.2.3	Compiler - 3.2.5
4.1.8.3 Shall be easily rehosted and retargeted	3.7.2.3	Compiler - 3.2.5
4.1.8.4 Process Ada source and produce an efficient, equivalent program	3.7.2.3	Compiler - 3.2.5
4.1.8.4.1 Process the complete Ada language	3.7.2.3	Compiler - 3.2.5

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.8.4.2 Design pragmas to support requirements, design language pragmas	3.7.2.3	Compiler - 3.2.4
4.1.8.5 Produce all necessary outputs required to implement separate compilation and linking and execution produce output listings any or all of which can be user suppressed.	3.7.2.3 3.7.2.4	Compiler - 3.2.5 Appendix C Appendix D Appendix E
4.1.8.5.1 Produce symbol attribute listing	3.7.2.3	Compiler - 3.2.5 3.3.14 Appendix E
4.1.8.5.2 Produce symbol cross reference listing		Compiler - 3.2.5 3.3.14 Appendix E
4.1.8.5.3 Produce source listings		Compiler - 3.2.5 3.3.2 Appendix E
4.1.8.5.4 Produce object program listing	3.7.2.3	Compiler - 3.2.5 3.3.13 Appendix E
4.1.8.5.5 Collect, store, and output source program and compilation statistics	3.7.2.3	Compiler - 3.2.5 3.3.4 Appendix E

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.8.5.6 Produce environment listing	3.7.2.3	Compiler - 3.2.5 3.3.13 Appendix E
4.1.8.5.7 Produce system management listings	3.7.2.3	Compiler - 3.2.5 Appendix E
4.1.8.6 Shall perform extensive error checking. Errors shall be associated with the source line number	3.7.2.3	Compiler - 3.3 3.3.2 3.3.4 Appendix E
4.1.8.6.1 Severities of compiler errors shall include	3.7.2.3	Compiler - Appendix E
4.1.8.6.2 Error messages shall contain an error identifier, severity code, and a descriptive text	3.7.2.3	Compiler - Appendix E
4.1.8.6.3 The compilers shall detect 100% of syntax errors and all semantic errors, any capacity requirement that has been exceeded; list of all error messages generated shall appear in the Users Manual.	3.7.2.3	Compiler - 3.2.5 Appendix E
4.1.8.7 Optimization shall occur at the user's option via language pragmas. Optimization with respect to memory usage and execution speed shall be provided.	3.7.2.3	Compiler - 3.2.4 3.2.7 3.2.8 3.2.9 3.2.10 3.2.11 3.2.12

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.8.8 Shall process Ada source at a rate of 1000 statements per minute or faster	3.2.1	Compiler - 3.2.5
4.1.8.9 Goal shall be no arbitrary limitations; clearly identify any limitations on internal capacities	3.7.2.3	Compiler - 3.5
4.1.9 LINKING and LOADING REQUIREMENTS facilities shall adhere to rules and specifications contained in language manuals	3.7.2.4	Linker - 3.2.5
4.1.10 Ada Program Library as specified in language manuals	3.7.2.4	Compiler - Appendix D
4.1.11 Project/Configuration Management facilities	3.7.2.2	CM - 3.2.5 KDBS - 3.2.5 3.3.1 3.3.5
4.1.11.1 Must provide the following reports: Configuration Composition Report Attribute Report Partition Report Attribute Select Report	3.7.2.1 3.7.2.2	KDBS - 3.2.5 3.3.1 3.3.2 CM - 3.2.5 ACLI - Command Utilities
4.1.11.2 Summary reports based on combinations of attribute, partition, configuration, or version qualifier	3.7.2.1 3.7.2.2	KDBS - 3.2.5 - 3.3.1 - 3.3.2 - 3.3.5 CM - 3.2.5 ACLI - Command Utilities

13

PHASE I SOW REQUIREMENTS	A - SPEC	B5 - SPEC
4.1.11.3 MAPSE shall include a mechanism for automatic stub generation. MAPSE shall store source code and maintain pertinent information for the stub	3.7.2	Compiler - 3.3.15 Linker - 3.2.5 3.3.2.3
4.1.12 High level I/O will be an extension of or alternative to package specified in the Ada Reference Manual	3.7.1.2 3.7.1.1 3.7.1.1 3.5.1.3	KDBS - 3.2.5.6 - 3.3.4
4.1.13 Specify and include in design terminal interface routines for batch and online keyboard terminals required for Phase II	3.1.5.3	KFW - 3.3.1 - 3.3.9 - 3.4.1.2.8
4.1.14 Identify, specify and design any additional host dependent programs necessary to implement MAPSE on IBM and Interdata computers	3.7.1.1 3.1.5.3	KFW - 3.2.5 - 3.3

Volume 3

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

APSE Command Language Interpreter

Prepared for

Rome Air Development Center
Griffiss Air Force Base, NY 13441

Contract No. F30602-80-C-0292

15

TABLE OF CONTENTS

16

TABLE OF CONTENTS (cont.)

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This document presents the Computer Program Development Specification (Type B5) for the Computer Program Configuration Item (CPCI) known as the Ada Programming Support Environment (APSE) Command Language Interpreter (ACLI). This specification establishes the performance, design, and test requirements for the ACLI.

1.2 FUNCTIONAL SUMMARY

The ACLI is a part of the MAPSE tool set and is responsible for interpreting the APSE Command Language (ACL). The ACLI is oriented for interactive use, and provides the user flexible, powerful capabilities for interacting with the process management and data base facilities of the KAPSE. ACL is a command programming language, combining features commonly found in both command languages and algorithmic programming languages. The Debugger executes as part of the ACLI, and allows users to monitor the execution of Ada programs interactively.

SECTION 2 - APPLICABLE DOCUMENTS

The following documents are applicable to this specification.

2.1 PROGRAM DEFINITION DOCUMENTS

1. Reference Manual for the Ada Programming Language, July 1980.
2. Requirements for Ada Programming Support Environment, STONEMAN, February 1980.
3. Statement of Work, Contract No. F30602-80-C-0292, 26 March 1980.

2.2 INTER-SUBSYSTEM SPECIFICATIONS

4. System Specification for the Ada Integrated Environment.
5. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.
6. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.
7. Volume 4, Computer Program Development Specification for CPCI APSE Configuration Management System.
8. Volume 8, Computer Program Development Specification for CPCI APSE Debugger.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

9. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.
10. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

11. Aho, A. V., and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vols. I and II, Prentice-Hall, 1972.
12. Fischer, C. N., D. R. Milton, and S. B. Quiring, Efficient LL(1) Error Correction and Recovery using only Insertions, Acta Informatica, 13, 1980.

20

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section provides the general description, identifies the external and internal interfaces, and presents the functional requirements and internal characteristics of the ACLI.

3.1.1 General Description

The ACLI is an Ada program that interprets the ACL (the ACL Reference Manual is supplied as Appendix A to this specification). ACL is designed for both interactive and batch use, and is both a programming language and a command language. As a programming language it provides string-valued variables, expressions, statements, control-flow constructs, and subprograms. As a command language it provides a flexible user interface to the process management and data base facilities of the Kernel APSE (KAPSE). In addition, the Debugger executes as part of the ACLI, enabling users to interactively monitor the execution of Ada programs. The specification for the Debugger is provided in [7].

The ACLI executes at the APSE process level (see [5]), and not as part of the KAPSE. The programming language constructs of ACL are interpreted directly within the ACLI. For the command language constructs, the ACLI issues calls, on behalf of the user, to the appropriate subprograms provided in the KDBS Utility Package, the Ada Run-time Support Package, and the KFW Interface Package ([5], [6]).

3.1.2 Peripheral Equipment Identification

The ACLI always reads from and writes to KAPSE data base objects. These objects may represent interactive terminals, host system files, or other host system devices. The ACLI must know whether it is connected to an interactive terminal in order to permit prompting. Otherwise, the ACLI has no knowledge of terminal characteristics, these are managed by the Ada Standard I/O Package and the low-level device handlers.

2/

3.1.3 Interface Identification

The ACLI uses the facilities of the Ada Standard I/O Package and the KDBS Utility Package provided by the KDBS [6], and of the KFW Interface Package provided by the KFW [5]. The Ada Standard I/O Package provides the functions required to perform Ada I/O, the KDBS Utility Package provides the functions required to manipulate data base objects and their attributes, and the KFW Interface Package provides the functions required to start and interact with other APSE processes.

The ACLI is initiated by the KFW Logon Process whenever that process performs a logon sequence for a user and the ACLI is specified in the user`s entry in the password file as the initial program. The ACLI may also be invoked by any APSE process. However, within the MAPSE tool set, only the Configuration Manager [7] and the ACLI itself invoke the ACLI. The Configuration Manager invokes the ACLI in order to process object derivation rules. The ACLI invokes a separate instance of itself in order to process command language subprograms.

Through the KFW Interface Package, the ACLI invokes other APSE programs. These include system tools as well as user programs. There is another set of facilities, called Command Utilities, which provide the user access to data base and process control functions at the command language level, and include facilities for report generation. The Command Utilities are distinct Ada subprograms, and are documented as part of the ACL Reference Manual. Those Command Utilities that may be included in command "pipelines" [Appendix A, Chapter 4] will execute as separate APSE processes. The Command Utilities that affect the execution environment of the ACLI will execute as part of the ACLI process.

3.2 FUNCTIONAL DESCRIPTION

This paragraph describes the functions of the ACLI, the program and equipment relationships and interfaces identified above, and the input/output utilization.

3.2.1 Equipment Description

The ACLI is designed to be highly portable, and will execute on both the IBM VM/370 and the Interdata 8/32. The particular character sets accepted by the ACLI may be extended to include host-dependent characters. Also host-dependent is a required user-activated mechanism, such as a break, that will effect a user-generated process interruption. However, the host-dependent characteristics of this mechanism will be handled by the terminal handlers and are unknown to the ACLI.

3.2.2 Computer Input/Output Utilization

The ACLI reads from and writes to KAPSE data base objects. As indicated in Paragraph 3.1.2, these objects may identify interactive terminals, host files, or other host devices. The interactive terminal interface is provided to support interactive users, the host file and host device interfaces are provided to support noninteractive batch (or background) execution of the ACLI. The only functional difference between interactive and batch execution of the ACLI is the prompting function, which is enabled for interactive use and disabled for batch use.

3.2.3 Computer Interface Block Diagram

See Figure 3-1.

3.2.4 Program Interfaces

The interfaces between the ACLI and the other system components include the invocation interface to invoke the ACLI from other APSE programs (including other MAPSE tools); the KAPSE virtual interface to access the underlying KAPSE facilities; and the Debugger Interface to call the Debugger, which is a subprogram of the ACLI.

3.2.4.1 Invocation Interface

On invocation, the ACLI is passed an ACL command string to interpret, the

25

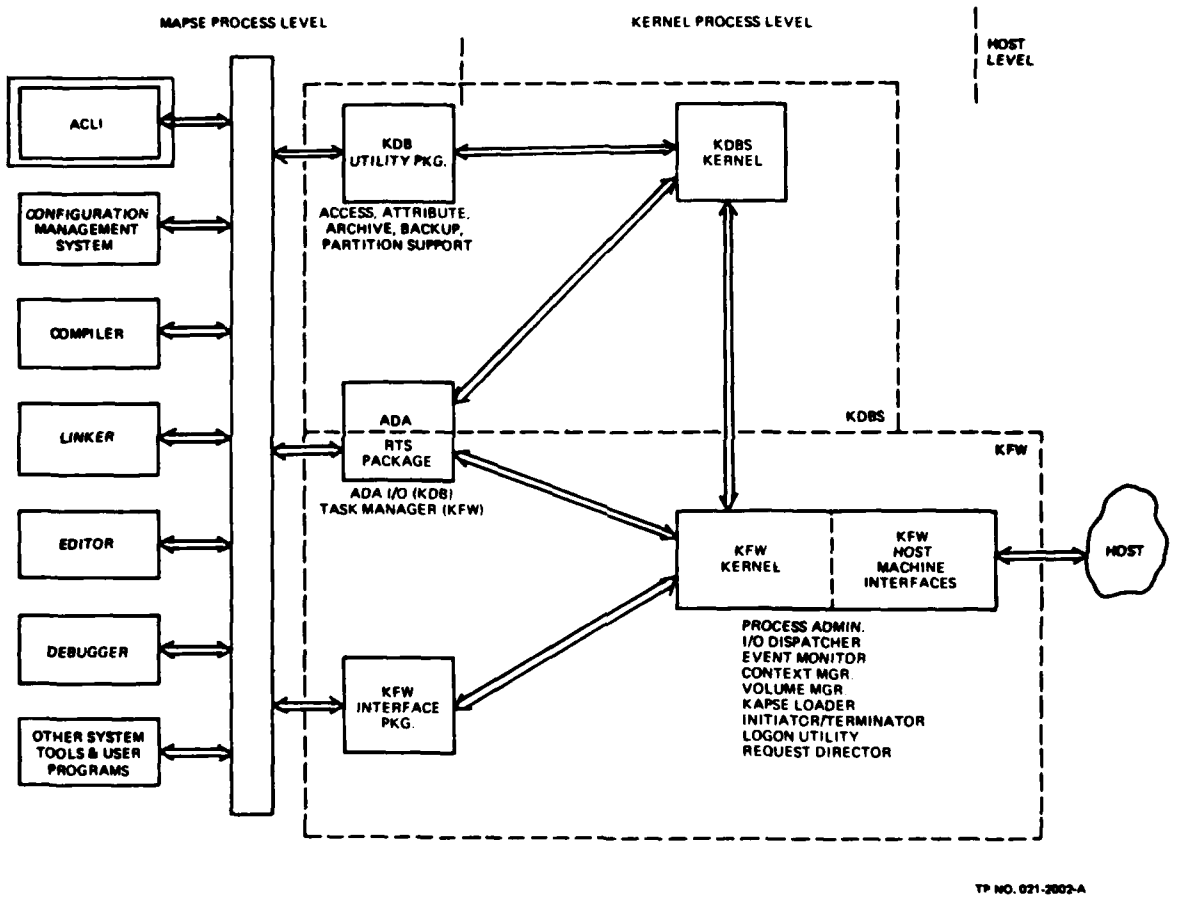


Figure 3-1. Interface Diagram

24

name of an ACL file to interpret, or neither, in which case the standard input file will be interpreted. The ACLI is specified as a function:

```
function ACLI ( OPTIONS       : in STRING := "";
                CMD_STRING    : in STRING := "";
                ACL_FILE NAME : in STRING := "";
              ) return RET_CODE;
```

The ACLI returns a value of type RET_CODE, which is an enumerated type defined in the KFW Interface Package.

The actual invocation of the ACLI is through the Start_Process or Overlay_Process functions, which are also defined in the KFW Interface Package:

```
Start_Process("ACLI",(arguments))
Overlay_Process("ACLI",(arguments))
```

The arguments may include: "OPTIONS=option-string, "CMD_STRING=cmd-string", and "ACL_FILE_NAME=acl-file-name". The latter two arguments may not both be specified.

Invocation of the ACLI is permitted from any MAPSE program. In addition the ACLI may be invoked by a KAPSE Framework (KFW) Logon Process whenever a logon sequence is performed for a terminal and the ACLI is specified as the initial program in the user`s entry in the password file.

The ACLI executes with both its real and effective user and group ids set by the invoking environment. Thus the ACLI enjoys no special privileges and executes as an ordinary APSE program.

3.2.4.2 KAPSE Virtual Interface

The KAPSE virtual interface consists of the KFW Interface Package, Ada Run-time Support Package, and KDBS Utility Package. The ACLI uses the full capabilities of the KFW Interface Package to create, suspend, and terminate processes, and to determine the status of active processes. The detailed specification of this package is provided in [5].

The Ada Run-time Support Package includes both tasking support and I/O support functions. The ACLI uses tasking support for its own internal tasking, and I/O support to perform Ada I/O. The KDBS Utility Package is used to manipulate KAPSE data base objects and their attributes. The detailed specification of these packages is provided in [6].

3.2.4.3 Debugger Interface

The Debugger executes as a subprogram of the ACLI and is invoked as a result of the ACLI encountering a Debugger directive:

```
Debug();              - to enter the Debugger subprogram
Debug(process_id);    - to enter the Debugger subprogram and
                      - indicate the process to debug
```

3.2.5 Function Description

The purpose of the ACLI is to interpret ACL commands. As indicated in Paragraph 3.2.4.1, ACL commands may be supplied as an argument to the invocation of the ACLI, in a KAPSE data base object named in the invocation of the ACLI, or most commonly from the standard input file. The semantic interpretation of ACL syntax is specified in the ACL Reference Manual. The remainder of the present specification will concentrate on the organization of the ACLI itself.

While the ACL provides a reasonably powerful programming language, it is especially geared toward providing a flexible command language for process handling. As a programming language, the ACLI interprets a variety of algorithmic programming contructs. As a command language, the ACLI calls on the KFW to create processes on behalf of the user, pass arguments to these processes, and receive out arguments and return values. The ACLI is responsible for setting the Standard Environment for invoked processes, permitting the Standard Input, Output, and Error files to be redirected according to the user's specifications. In addition, a number of Command Utilities are provided to interact with active processes.

The ACLI operates as a conventional language interpreter. The functionality of the ACLI is factored among three phases: lexical analysis (Scanner), syntactic analysis (Parser), and interpretation (Interpreter). The data flow structure of these phases is provided in Figure 3-2. These phases will be detailed in Paragraph 3.3 below.

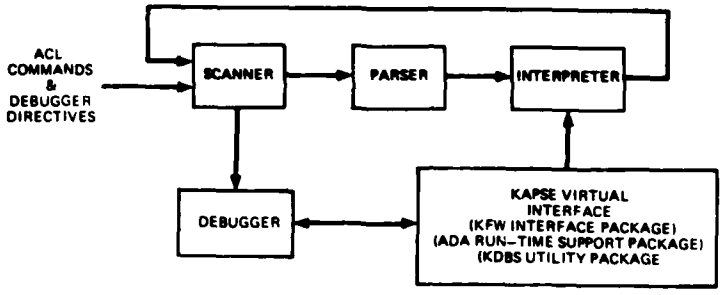


Figure 3-2. ACLI Phases

3.3 DETAILED FUNCTIONAL REQUIREMENTS

Functional requirements are provided for the Scanner, Parser, and Interpreter.

3.3.1 Scanner

The Scanner, using Ada standard I/O, reads ACL commands from either the Standard Input File, an ACL subprogram object, or from a user profile object. The Scanner transforms these commands into a sequence of lexical tokens. In addition, the Scanner is responsible for fielding attention interrupts that may be sent by an interactive user.

3.3.1.1 Inputs

Although it may be invoked by other APSE programs or system tools (the Configuration Manager, Configure, in particular), the ACLI is usually invoked by the KFW Logon Process. In the usual case, the ACLI is invoked without the CMD_STRING or ACL_FILE_NAME arguments being supplied, so input

is taken from the standard input file that is set in the Standard Environment of the ACLI process. This file will normally be a terminal, in which case the ACLI will execute interactively. If arguments are supplied to the ACLI, commands will be read from the CMD_STRING or the data base object named by ACL_FILE_NAME. In this case, the ACLI will not interact with the user.

The Scanner must also process input arising from command substitutions that are generated by the Interpreter. Such input will be inserted by the Interpreter at the appropriate point in the normal input (see Paragraph 4.3, ACL Reference Manual).

The input must have the ACL syntax specified in the ACL Reference Manual. The Scanner is responsible for detecting lexical errors, the Parser for detecting syntactic errors.

3.3.1.2 Processing

Lexical analysis is performed by a simple, table-driven, finite-state recognizer. Algorithms for such recognizers are well known and simple to implement.

The Scanner also provides a task entry for the attention interrupt that may be directed by the user to the ACLI process. Upon receipt of the interrupt, any child processes will be suspended, and any ACL loops or inline subprograms will be exited. If the ACLI is executing interactively, it will prompt and be prepared to receive input from the terminal. If not, the ACLI will terminate.

Because the Scanner and Parser operate logically as coroutines, each is to be defined as an Ada task. Thus the Scanner performs as a producer task, generating token pairs for a consumer task, the Parser. The Scanner must also recognize which tokens constitute Debugger directives, these tokens will be parsed separately by the Debugger task.

3.3.1.3 Output

The output of the Scanner is a stream of (token number, value) pairs. The token number is of enumeration type, and identifies the class of the token. The value is the character string that was the source representation of the token.

In interactive mode, the Scanner will also issue a prompt string to the user`s terminal whenever the Scanner is prepared to accept input. The prompt string will be taken from the environment variable $PROMPT.

3.3.2 Parser

The Parser performs a syntax-directed translation from the stream of (token, value) pairs supplied by the Scanner into a syntax tree suitable for interpretation by the Interpreter.

3.3.2.1 Inputs

The input to the Parser is a stream of (token, value) pairs. As indicated above, the Scanner and the Parser operate as producer and consumer tasks respectively.

3.3.2.2 Processing

Parsing will be accomplished with the Strong LL(1) algorithm. The LL(1) technique provides small, efficient, table-driven parsers. In addition, the FMQ error-recovery algorithm will be used to generate high-quality error messages for syntactic errors.

The Parser will prepare a syntax tree on a per-statement basis. That is, a tree is sent to the Interpreter for interpretation only when the tree comprises a complete statement. Note that this implies that compound statements (if, case, and loop statements) must be completely parsed before they are interpreted.

There is one significant exception to the above rule. Substitutes (see Appendix A) are parsed and interpreted as soon as they are encountered in the input stream. Thus a substitute appearing in a loop statement will be evaluated exactly once, regardless of the number of times the loop is traversed.

3.3.2.3 Outputs

The output of the parser is a sequence of syntax trees. Each tree represents a complete statement (or substitute, see above) in a form suitable for interpretation by the Interpreter. The detailed design of the tree format will be provided in the C-5 Specification.

3.3.3 Interpreter

The Interpreter interprets, or "executes", the syntax trees provided by the parser. The interpretation of each syntactic construct is fully specified in the ACL Reference Manual.

3.3.3.1 Inputs

The input to the Interpreter is a syntax tree generated by the Parser. The Interpreter is invoked once per tree.

3.3.3.2 Processing

The Interpreter traverses the syntax tree, and performs operations corresponding to the traversed syntactic constructs. These operations are detailed in the ACL Reference Manual.

The Interpreter maintains a symbol table containing the values for all command language variables. The symbol table is "flat", because there are no scoping rules restricting the visibility of variables in ACL. Variables are declared when they are used, and they may possess only two kinds of values: strings, and aggregates of strings.

3.3.3.3 Outputs

The chief outputs of the Interpreter consist of calls through the KAPSE virtual interface to the process management functions of the KFW and the data base functions of the KDBS. The Interpreter also manipulates the input buffers of the Scanner to insert the values returned by command substitution.

3.3.4 Command Utilities

The Command Utilities are logically separate programs. They may be implemented either in Ada or in ACL. Those utilities that affect the execution state of the ACLI will be implemented within the ACLI as

subprograms. All other Utilities may appear in command pipelines and are invoked as separate processes.

The Command Utilities make available to the ACLI user the applicable functionality provided by the KFW Interface Package and the KDBS Utility Package. Of particular interest are the Find Object and Read Attribute Value Utilities, which can be combined to generate reports.

3.4 ADAPTATION

This section describes any adaptation that might be required to rehost the ACLI.

3.4.1 General Environment

On some hosts, the user interface of the ACLI will be limited to half-duplex.

3.4.2 System Parameters

The number of processes that a single user can have active may be limited on particular hosts.

3.4.3 System Capacities

The memory allocated to the ACLI on particular hosts may limit the size and depth of nesting permitted for compound statements in ACL.

3.5 CAPACITY

Not applicable.

SECTION 4 - QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the APSE Command Language Interpreter (ACLI). The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the ACLI. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government-approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data is processed. For example, a review of hardcopy test data might be used to verify that the values of specific parameters are correctly computed.

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

4.1.1 Subprogram Testing

Following unit testing, individual modules of the ACLI shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2 Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

CPCI testing shall be performed on all development software of the ACLI. This specification presents the performance criteria that the developed CPCI must satisfy. The correct performance of the ACLI will be verified by testing its major functions. Successful completion of the program testing

Figure 4-1. Test Requirements Matrix

SECTION	TITLE	INSP.	ANAL.	DEMO.	DATA.	SECTION NO.
3.3.1	Scanner		X		X	4.2.1, 4.2.2
3.3.2	Parser		X		X	4.2.1,4.2.2
3.3.3	Interpreter				X	4.2.2
3.3.4	Command Utilities				X	4.2.2

that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3 System Integration Testing

System integration testing involves verification of the integration of the ACLI with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the ACLI performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details

regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

The Scanner and Parser shall be validated both by analysis of the constituent algorithms and by formal review of test data. The Interpreter and the Command Utilities shall be validated by formal review of test data.

In order to provide for program testing of the ACLI in advance of the completion of the KFW and KDBS, a facility shall be provided to log rather than execute all calls made on KFW and KDBS functions. This logging facility shall be used to generate test output data from the test input scripts. The test input scripts are a subset of those that will constitute the ACL Validation Test Set described below in Paragraph 4.3. Validation shall be by formal review of test data.

The ACL Validation Test Set shall be augmented with tests for validating each interface described in Paragraph 3.2.4. A program shall be written to execute the augmented test set automatically. Validation of the system integration of the ACLI shall be by formal review of test data.

4.2.1 Analysis of Algorithms

The coded scanning and parsing subprograms shall be inspected to ensure that they conform to provably correct algorithms that have appeared in the literature (e.g., [11]).

4.2.2 Formal Review of Test Data

Drivers shall be written to generate input data and to log output data. Test input scripts and expected test output shall be developed by test personnel in accordance with subprogram and program specifications. Testing shall consist of comparing expected output data with test output data. To minimize the effort required for developing drivers, a bootstrap approach shall be taken. Thus an initial driver for the Scanner shall be provided, the Scanner shall serve as the driver for the Parser, and the Scanner-Parser shall serve as the driver for the Interpreter. Because the Command

Utilities require less complex drivers, independent drivers shall be provided for them in order to facilitate parallel development.

4.3 ACCEPTANCE TESTING

An ACL Validation Test Set shall be prepared, consisting of a comprehensive set of test scripts along with expected output for each script. Scripts shall be provided to test all forms of each ACL construct specified in the ACL Reference Manual (see Appendix A). Scripts shall be provided to test each Command Utility separately, and where feasible, in combination. Scripts shall be provided to test the invocation of the ACLI by the Configuration Manager.

A program shall be written that will automatically execute the ACL Validation Test Set and compare the expected output data with the test output data. The output of this program will be a report documenting the successful or unsuccessful execution of each script in the Test Set. Successful execution of the entire Test Set shall constitute successful completion of the acceptance test. The Test Set shall be included as part of the Computer Program Test Procedures to be approved by the Government.

SECTION 5 - DOCUMENTATION

5.1 GENERAL

The documents that will be produced during the implementation phase in association with the ACLI are:

1. Computer Program Development Specification (Type B5) - Update
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. User`s Manual
6. Rehostability Manual
7. ACL Reference Manual
8. MAPSE Tools Reference Handbook
9. ACL Validation Test Set Manual

5.1.1 Computer Program Development Specification

The final ACLI B5 Specification will be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II.

5.1.2 Computer Program Product Specification

A Type C5 Specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document will be used to specify the design of the ACLI and the development approach implementing the B5 specification. This document will provide the detailed description that will be used as the baseline for any Engineering Change Proposals.

5.1.3 Computer Program Listings

Listings that result from the final compilation of the accepted ACLI will be delivered. Each compilation unit listing will contain the corresponding source, cross-reference, and compilation summary. The source listing will contain the source lines from any included source objects.

5.1.4 Maintenance Manual

An ACLI Maintenance Manual will be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the ACLI to be easily maintained by personnel other than the developers. The documentation will be structured to relate quickly to program source. The procedures required for debugging and correcting the ACLI, along with debugging aids that have been incorporated as integral parts of the ACLI, will be described and illustrated. Sample scripts for compiling ACLI components, for relinking the ACLI in parts or as a whole, and for installing new releases will be supplied.

5.1.5 User`s Manual

A User`s Manual shall be prepared in accordance with DI-M-30421, and will contain all information necessary for the operation of the ACLI. Because of the virtual user interface persented by the ACLI, a single manual is sufficient for all host computers. Information relevant to specific hosts will be contained in an appendix. A complete list of all ACLI diagnostic messages will be included with supplemental information supplied to assist the user in locating and correcting ACL errors.

5.1.6 Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual will be prepared to describe step-by-step procedures for rehosting the ACLI on a different computer.

5.1.7 ACL Reference Manual

A Reference Manual distinct from the User`s Manual shall be prepared to define the syntax and semantics of ACL. The Reference Manual will adopt the form and level of presentation that were used in the Ada Reference Manual.

5.1.8 MAPSE Tools Reference Handbook

A MAPSE Tools Reference Handbook shall be produced containing syntax diagrams for all command constructs.

5.1.9 ACL Validation Test Set Manual

A manual shall be provided that describes the procedures for running the ACL Validation Test Set. A machine-readable form of the ACL Test Set shall be provided along with listings of each member of the Test Set.

APPENDIX A

APSE COMMAND LANGUAGE

REFERENCE MANUAL

Vol 3

SECTION 1 - INTRODUCTION

The APSE Command Language (ACL) is a command programming language that provides an interface to the MAPSE system. The primary function of ACL is to provide a convenient user interface to the process-related facilities of the MAPSE. ACL is processed by the ACL Interpreter (ACLI), which is a MAPSE program. ACL contains a variety of Ada-like control-flow constructs, and provides variables and parameter passing mechanisms. Thus ACL is itself a programming language, but one that is specifically designed for process management.

The ACLI is used almost exclusively to invoke other MAPSE programs. Many of the commands are directives to the ACLI to initiate a named program as a MAPSE process. A few commands are provided to alter the state of the ACLI itself; these are executed entirely within the ACLI process. In addition, the ACLI may, for efficiency, execute directly some commands that could also have been implemented as separate programs.

SECTION 2 - LEXICAL ELEMENTS

2.1 CHARACTER SET

All ACL constructs may be represented with a basic graphic character set, which is identical to the set used for Ada:

(a) upper case letters

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

(b) digits

0 1 2 3 4 5 6 7 8 9

(c) special characters

" # % & ' ( ) * , - . / : ; < = > _ |

(d) the space_character

The character set may be extended to include further characters from the 95 character ASCII graphics set:

(e) lower case letters

a b c d e f g h i j k l m n o p q r s t u v w x y z

(f) other special characters

! $? @ [\ ] ^ ` { } ~

Except within character strings, any lower case letter is equivalent to the corresponding upper case letter. A rule will be provided below for transliterating unavailable special characters within strings.

2.2 LEXICAL UNITS

Lexical units in ACL are expressed in an essentially free format. A sequence of non-special characters (i.e, letters or digits) forms a word. Words may be delimited by special characters (except for the underscore, which is not a delimiter), space_characters, or horizontal tabs. A word may also be a character string, which may contain escaped special characters (see Paragraph 2.5). Each word must fit on a line, thus an end-of-line also delimits otherwise contiguous words.

The following compound symbols are also delimiters:

```
=>  ..  :=  /=  >=  <=  ->  ->>  #>  #>>  '<  >'
```

2.3 IDENTIFIERS

Identifiers are used as reserved words, variables, and names. An identifier is also a character string (see Paragraph 2.5). The apparently ambiguous uses of identifiers are easily distinguishable in context.

```
identifier  ::=  letter { [ underscore ] letter_or_digit }
letter_or_digit  ::=  letter  |  digit
letter  ::=  upper_case_letter  |  lower_case_letter
```

Note that identifiers differing only in the use of corresponding upper- and lower-case letters are considered to be identical. Underscores are treated as part of the identifier, and are therefore significant.

Examples:

```
X          X15        EXFLAG          case
Exflag     exflag     ex_flag         Ada
```


4B

2.4 NUMBERS

Numbers, or numeric literals are integers to which may be appended an optional base. Numbers are also a special case of character strings (see 2.5).

```
number ::= decimal_number | based_number
decimal_number ::= digit { [ underscore ] digit }
based_number ::= base # based_integer
base ::= decimal_number
based_integer ::=
        extended_digit { [ underscore ] extended_digit }
extended_digit ::= digit | letter
```

Isolated underscores may be inserted between adjacent digits to provide readability, but they are not otherwise significant. For bases greater than ten, the extended digits include the letters A through F, signifying 10 through 15 respectively.

Examples:

```
0          17          32_768       -- decimal integers
2#101111   2#10_1111   16#2F        -- integers with value 47
```

2.5 CHARACTER STRINGS AND QUOTING

A character string is a sequence of zero or more characters.

```
character_string ::= " { character } " | character { character }
```

"" alone denotes the empty string. Character strings may contain special characters that otherwise would possess a special meaning. Enclosing a character string within double quotes (") prevents this special meaning from being applied. In order to include the double quote itself within a string, it must be written twice. Character strings that would otherwise be construed as reserved words (see Paragraph 2.7) must also be quoted.

44

Examples:

```
""                                  -- empty string
anystring                           -- quoting not required
a        " "      "a"      """"    -- four strings of length one
"a longer string"                   -- quoting required to preserve blanks
"if"                                -- quoting prevents interpretation as
                                    -- a reserved word
```

2.6 COMMENTS

A comment begins with two hyphens and is terminated by the end of the line. Comments have no effect on the interpretation of ACL, and are inserted solely for human readability.

Examples:

```
-- this is a simple comment
if -- the hyphens and these words have no effect

-- comments split over two lines
-- require additional hyphens

---- the first two hyphens start the comment
```

2.7 RESERVED WORDS

The identifiers listed below have a special meaning within the language and are reserved.

and	begin	case	do
else	elsif	end	exit
for	function	if	in
loop	mod	not	null
or	others	out	procedure
rem	renames	return	then
when	while	xor	

2.8 VARIABLES AND THE SUBSTITUTION RULE

A '%' character is used to distinguish ACL variables from other words.

variable ::= % identifier

Variables appear in either a defining or an applied context. A defining context is one that sets the value of a variable, while an applied context is one that uses the value. The substitutionrule is fundamental to the interpretation of ACL. The rule is that the ACLI, upon encountering a variable in an applied context, replaces it with its value. This replacement can be considered to be performed simultaneously for all such variables in a statement.

Examples:

%Exflag %a %X_1

2.9 OBJECT NAMES AND NAME EXPANSION

Names are provided to identify objects in the KAPSE Data Base.

```
name ::= directory [ xidentifier ] [ ' category ] [ qualifier ]
directory ::= [ / ] { [ xidentifier / ] }
category ::= xidentifier
qualifier ::= . branch [ . version ]
branch ::= xidentifier
version ::= * | decimal_number
xidentifier ::= xletter { xletter | digit }
xletter ::= * | letter
```

The directory is optional. If absent, the current directory is prepended to the name. The qualifier is also optional. If absent, the current branch is appended to the name if the named object is an abstract object.

If a "*" appears in an object name, the name represents a pattern that is matched against Data Base object names. "*" matches any sequence (including the empty sequence) of characters that may appear in an object, branch, or version name. Note that "/", "'", and "." are not matched, since these characters delimit the components of a name. The list of matched names is sorted into lexicographic order and formed into an aggregate (see Paragraph 3.1.1).

Examples:

```
/USAF/RADAR/Track          -- absolute pathname
Track/finder               -- current directory will be prepended
                           -- to this
finder'XQT                 -- with category qualifier
finder'XQT.VM370           -- with category and branch qualifier
finder'XQT.VM370.6         -- with category, branch, and version
                           -- qualifier
finder.VM370               -- with only branch qualifier
Track/*                    -- all objects in partition "Track"
Track/f*                   -- all objects in partition "Track"
                           -- whose names begin with "f"
```

SECTION 3 - VARIABLES, TYPES, AND EXPRESSIONS

3.1 VARIABLES AND TYPES

As indicated in Section 2, all non-reserved identifiers are either variables or object names, with variables being distinguished by an initial '%'. Variables are declared and implicitly typed when they are first used. ACL is essentially typeless, allowing only strings and simple aggregates.

3.1.1 Aggregates

An aggregate is a list of character_strings.

```
aggregate ::=
        ( character_string { [ , ] character_string } )
```

Note that the components of an aggregate may be separated by either commas or space_characters.

Examples:

```
( 1, 2, 3 )
( abc, def, ghi )
( "f", "then", "else" )
( A, 8#40 )
```

3.1.2 Sliced Variables

A sliced variable denotes a sequence of consecutive components within a variable that has a string or an aggregate value.

```
sliced_variable  ::= variable ( discrete_range )
discrete_range   ::= simple_expression .. simple_expression
                   | simple_expression
```


48

The substitution rule replaces each sliced variable appearing in an applied context with its value. The value of a sliced variable is an aggregate if the range spans more than one component of an aggregate variable. If the variable is a string, slices are used to denote substrings. If the range is a single simple_expression, then the slice denotes a single character of a string variable, or a single component of an aggregate variable.

Examples:

```
%Plist(2..3)
%Xstring(1..19)
%stat_array(14)
```

3.2 EXPRESSIONS

An expression is a formula that defines the computation of a value.

```
expression ::=
        relation { and relation }
        | relation { or relation }
        | relation { xor relation }
        | relation { and then relation }
        | relation { or else relation }
relation ::=
    simple_expression [ relational_operator simple_expression ]
    | simple_expression [ not ] in aggregate
simple_expression ::=
    [ unary_operator ] term { adding_operator term }
term ::= primary { multiplying_operator primary }
primary ::=
    character_string | number | ( expression )
    | aggregate | variable | sliced_variable | pipeline
```

Since ACL is essentially a typeless language, overloading of operators is not needed. Each operator will convert its operands, if necessary and if possible, to conform to the required operation. Thus, for example, the string "15" may appear as an operand to the string catenation operator as well as to an adding_operator.

Variables and pipelines (i.e, program invocations, see Paragraph 4.2) may syntactically appear as primaries. However, variables are replaced by their values and pipelines are executed for their values before any operator may be applied to them.

Examples of primaries:

```
blue                              -- character string
72                                -- number
(%Status + 1)                     -- parenthesized expression
(blue,green)                      -- aggregate
%Status                           -- variable
%Proc_list(1..4)                  -- sliced variable
echo(%Proc_list)  |  sort()       -- pipeline
```

Examples of expressions:

```
blue                              -- primary
%Status * 2                       -- term
-%Status                          -- simple expression
%Status + 4                       -- simple expression
abc & def                         -- simple expression
%Status > 0                       -- relation
blue in %Proc_list                -- relation
%Status and (2 not in %Proc_list) -- expression
(%Status + 1) * 2                 -- expression
```

3.3 OPERATORS AND PRECEDENCE

The following operators are arranged in order of increasing precedence.

```
logical_operator  ::= and | or | xor
relational_operator  ::= = | /= | < | <= | > | >=
adding_operator  ::= + | - | &
unary_operator  ::= + | - | not
multiplying_operator  ::= * | / | mod | rem
```


3C

The short circuit control forms "and then" and "or else" have the same precedence as logical operators. The membership tests "in" and "not in" have the same precedence as relational operators.

All operands of a factor, term, simple expression, or relation, and the operands of an expression that does not contain a short circuit control form, are evaluated (from left to right) before application of the corresponding operator. Operand evaluation includes the execution of a function call appearing as a primary. The right operand of a short circuit control form is evaluated if and only if the left operand has a certain value (see Paragraph 3.3.1).

The preceding implies that within an expression, operators of higher precedence are applied first, and within precedence levels operators are applied in textual order from left to right.

Examples:

```
%A * %B + %C                    -- same as (%A * %B) + %C
%A < 3 and 4 = %B               -- same as (%A < 3) and (4 = %B)
not %A or %B                    -- same as (not %A) or %B
```

3.3.1 Logical Operators and Short Circuit Control Forms

The logical operators are applicable to Boolean values.

Operator	Operation
and	conjunction
or	inclusive disjunction
xor	exclusive disjunction

Each logical operator attempts to interpret its operands as Boolean values. The only operands that are interpreted as FALSE are the string "0" and the aggregate with the string "0" as its only component. All other operands are interpreted as TRUE (except for aggregates). Each logical operator produces as its result either the string "1" for TRUE, the string "0" for FALSE, or an aggregate of these values.

If both operands are aggregates, the operator is performed on matching components, and the result is an aggregate as long as the longest operand. The shorter operand is extended, before the operator is applied, with sufficient FALSE values to match the length of the longer operand.

The short circuit control forms "and then" and "or else" are provided to avoid the evaluation of the right operand in certain situations. If the left operand of "and then" evaluates to FALSE, the right operand is not evaluated and the value of the expression is FALSE. If the left operand of "or else" evaluates to TRUE, the right operand is not evaluated and the value of the expression is TRUE. If both operands are evaluated, "and then" delivers the same result as "and" and "or else" delivers the same result as "or". If applied to operands that are both aggregates, the short circuit control forms will not prevent the evaluation of the right operand.

Examples:

```
%A or %B
%B and then (Blue in %A)
```

3.3.2 Relational and Membership Operators

The relational and membership operators operate on character strings and aggregates and produce a Boolean value.

Operator	Operation
= , /=	equality, inequality
< , <= , > , >=	test for ordering
in , not in	membership

The relational operators may be applied to operands that are both strings or both aggregates. Strings are compared lexicographically according to the ASCII character set. For strings that could be interpreted as numbers, this comparison produces the same result that would be generated by a numeric comparison, except when leading zeros are present. Thus leading zeros are significant.

Two aggregates are equal if and only if their corresponding components are equal. Aggregates of different length are therefore never equal. An aggregate appearing as an operand to an ordering operator is treated as a string that is the catenation of its components.

The membership test operators require a string as the left operand and an aggregate as the right operand. The test is whether the left operand appears as a component of the right operand.

Examples:

```
Blue = %Status
ERROR_3 in %Status_list(1..4)
%A > 17
```

3.3.3 Adding Operators

Operator	Operation
+	addition
-	subtraction
&	catenation

Addition and subtraction are only defined for non-aggregate operands that can be interpreted as numbers (see 2.4). Catenation accepts either two strings or two aggregates as operands, and the result is of the same form.

Examples:

```
2 + 17
%Status - 4
%A & "short string"
Blue & Green
```

If both operands are aggregates, the operator is performed on matching components, and the result is an aggregate as long as the longest operand. The shorter operand is extended, before the operator is applied, with sufficient FALSE values to match the length of the longer operand.

The short circuit control forms "and then" and "or else" are provided to avoid the evaluation of the right operand in certain situations. If the left operand of "and then" evaluates to FALSE, the right operand is not evaluated and the value of the expression is FALSE. If the left operand of "or else" evaluates to TRUE, the right operand is not evaluated and the value of the expression is TRUE. If both operands are evaluated, "and then" delivers the same result as "and" and "or else" delivers the same result as "or". If applied to operands that are both aggregates, the short circuit control forms will not prevent the evaluation of the right operand.

Examples:

```
%A or %B
%B and then (Blue in %A)
```

3.3.2 Relational and Membership Operators

The relational and membership operators operate on character strings and aggregates and produce a Boolean value.

Operator	Operation
= , /=	equality, inequality
< , <= , > , >=	test for ordering
in , not in	membership

The relational operators may be applied to operands that are both strings or both aggregates. Strings are compared lexicographically according to the ASCII character set. For strings that could be interpreted as numbers, this comparison produces the same result that would be generated by a numeric comparison, except when leading zeros are present. Thus leading zeros are significant.

5-2

3.3.4 Unary Operators

The unary operators are applied to a single operand.

Operator	Operation
+	identity
-	negation
not	logical negation

The unary "+" and "-" are applicable only to non-aggregate operands that can be interpreted as numbers (see Paragraph 2.4). The operator "not" when applied to a non-aggregate produces the string with the Boolean value opposite to that of the operand. However only the canonical strings "1" and "0" are produced. Applying "not" to an aggregate produces an aggregate of equal length with each component possessing the Boolean value opposite to that of the corresponding operand component.

Examples:

```
+ 4
- %Status
not (Blue = %Status)
```

3.3.5 Multiplying Operators

The multiplying operators are defined only for non-aggregate operands that can be interpreted as numbers

Operator	Operation
*	multiplication
/	division
mod	modulus
rem	remainder

Two aggregates are equal if and only if their corresponding components are equal. Aggregates of different length are therefore never equal. An aggregate appearing as an operand to an ordering operator is treated as a string that is the catenation of its components.

The membership test operators require a string as the left operand and an aggregate as the right operand. The test is whether the left operand appears as a component of the right operand.

Examples:

```
Blue = %Status
ERROR_3 in %Status_list(1..4)
%A > 17
```

3.3.3 Adding Operators

Operator	Operation
+	addition
-	subtraction
&	catenation

Addition and subtraction are only defined for non-aggregate operands that can be interpreted as numbers (see 2.4). Catenation accepts either two strings or two aggregates as operands, and the result is of the same form.

Examples:

```
2 + 17
%Status - 4
%A & "short string"
Blue & Green
```

3.3.4 Unary Operators

The unary operators are applied to a single operand.

Operator	Operation
+	identity
-	negation
not	logical negation

The unary "+" and "-" are applicable only to non-aggregate operands that can be interpreted as numbers (see Paragraph 2.4). The operator "not" when applied to a non-aggregate produces the string with the Boolean value opposite to that of the operand. However only the canonical strings "1" and "0" are produced. Applying "not" to an aggregate produces an aggregate of equal length with each component possessing the Boolean value opposite to that of the corresponding operand component.

Examples:

```
+ 4
- %Status
not (Blue = %Status)
```

3.3.5 Multiplying Operators

The multiplying operators are defined only for non-aggregate operands that can be interpreted as numbers

Operator	Operation
*	multiplication
/	division
mod	modulus
rem	remainder

Division and remainder are defined by the relation

A = (A/B)*B + (A rem B)

where (A rem B) has the sign of A and an absolute value less than the absolute value of B. Division also satisfies:

(-A)/B = -(A/B) = A/(-B)

The result of the modulus operation is such that (A mod B) has the sign of B and an absolute value less than the absolute value of B; in addition, this result must satisfy the relation:

A = B*N + (A mod B)

for some integer value of N.

Examples:

```
2 * %Status
%Proc_id_list(1)/5
%A mod 3
%B rem 7
```

SECTION 4 - PROCESS INVOCATION

4.1 PROCESS CALLS

```
process_call  ::=  program_name ( [ actual_parameter_part ] )
program_name  ::=  name
actual_parameter_part  ::=
          parameter_association { , parameter_association }
parameter_association  ::=
          [ formal_parameter => ] actual_parameter
formal_parameter  ::=  identifier
actual_parameter  ::=
     expression  |  variable  |  sliced_variable
```

The program_name must name an Ada program object (i.e, fully compiled and linked), or and ACL program object

A process call is the mechanism that causes the execution of an Ada program as a MAPSE process. When interpreting a process call, the ACLI, in order to correctly interpret the actual parameters, first accesses the symbol table associated with the named program. The arguments are then evaluated, and the process is initiated with part of its standard environment having been initialized to correspond to the invoking context (see Paragraph 4.2).

A process call may return a value. For Ada programs, the type of the value returned is specified in the specification of the program name. The ACLI will attempt to format the value returned into a character string or an aggregate of character strings. This return value is not to be confused with the standard output of the process. Ordinarily, return values are used to return completion status codes to the ACLI. For processes that terminate abnormally, exception names may be returned as the "return value". For processes that do not return values, the ACLI will provide a return code of "0" for any process that terminates normally, and "1" for any process that terminates abnormally and does not return an exception name.

57

If the program_name names an ACL program instead of an Ada program, a separate instance of the ACLI is created as a separate process (except for "inline" invocation, see Paragraph 5.9). This process will take commands from the named ACL program. Command language procedures will be fully discussed in Section 6.

For both Ada and ACL programs, "in", "out", and "in out" parameters are permitted. The form of the parameter is determined by the specification in the called program. For "out" and "in out" parameters, the actual argument must be a variable or sliced_variable.

Examples:

```
Ada(Track/prog'txt)
ACLI(CMD_STRING=>"echo(%Proc_list)")
ACLI(TRACK/Lprog'CMD)
LINK((Aprog,Bprog),PROG'XQT)
```

4.2 PIPELINES

A pipeline establishes the invoking context of a process

```
pipeline ::=
    [ input_redirection ] process_list [ output_redirection ]
process_list ::=
    process_call [ error_redirection ]
        {|process_call [ error_redirection ] }
input_redirection ::= -> object_name
output_redirection ::= -> object_name | ->> object_name
error_redirection ::= *> object_name | *>> object_name
```

Input redirection indicates that the standard input for the immediately following process call is to be taken from the named object. If input redirection is not specified, the standard input is the same as that for the invoking ACLI process. Output redirection indicates that the standard output for the immediately preceding process call is to be written to the named object. If output redirection is not specified, standard output is

51

the same as that for the invoking ACLI process. There are two forms of output redirection: "->" indicates that the named object is to be truncated to zero length before writing, "->>" indicates that the output is to be appended to the named object. In both cases, the object is created if it does not exist.

Except for the initial process, each process specified in a process call list takes its standard input from the standard output of the previous process. Each process may additionally have its standard error redirected via an error redirection. Error redirection is similar to output redirection, with "#>" indicated initial truncation of the object and "#>>" indicating append mode. Processes within a list can be considered to execute logically in parallel.

The value returned by a pipeline is the value returned by the last process in the pipeline.

Examples:

```
list(Radar/Track) | sort()
list(Radar/Track) | sort() -> slist
list(Radar/Track) #>> saved_errors | sort()
slist -> copy(Radar/Track/Savelist)
```

4.3 COMMAND SUBSTITUTION

A pipeline can be invoked so as to return its standard output into the command stream.

```
substitute ::= # pipeline # | '# pipeline #'
```

With the non-quoted form, the ACLI will reformat the standard output of the pipeline into a character string or an aggregate. The character string must be an identifier or a number, the aggregate must contain only identifiers or numbers. In the reformatting, sequences of spaces, horizontal tabs, and newlines serve only to delimit the identifiers and numbers. This form of the substitute may appear in the place of any identifier, number, or aggregate.

58

The quoted form of substitute preserves the standard output as a quoted character string, and may appear in place of a quoted character string.

Command substitution is performed for each substitute appearing in a simple statement (see Paragraph 5.1) before the statement is interpreted. Command substitution is done before object name expansion (see Paragraph 2.9).

The substitute construct belongs properly to the "metasyntax" of ACL. Thus "substitute" will not appear elsewhere in the syntactic description.

Examples:

```
# list(Track) #          -- returns an aggregate consisting of the
                         -- members of partition "Track"
'# sort(old_list'TXT) #' -- returns a quoted string containing the sorted
                         -- file
```

SECTION 5 - STATEMENTS

A statement is the fundamental unit of interpretation for the ACLI. That is, no portion of a statement is interpreted (except for command substitution, see Paragraph 4.3)) until the entire statement is read by the ACLI.

5.1 SIMPLE AND COMPOUND STATEMENTS - SEQUENCES OF STATEMENTS

A statement may be simple or compound. A simple statement contains no other statements. A compound statement may contain simple statements or compound statements.

```
sequence_of_statements  ::=  statement { statement }
statement  ::=
     simple_statement
     | compound_statement
simple_statement  ::=
     null_statement
     | assignment_statement
     | pipeline_statement
     | exit_statement
     | return_statement
     | inline_statement
     | renames_statement
compound_statement  ::=
     if_statement
     | case_statement
     | loop_statement
null_statement  ::=  null ;
```

Statements in a sequence are interpreted in succession. No portion of a statement is interpreted until the entire statement is read by the ACLI. Thus statements within a compound statement are not interpreted until the

60

end of the compound statement is reached.

Interpretation of a null statement has no other effect than to pass on to the next action.

5.2 ASSIGNMENT STATEMENT

An assignment statement defines the value of a variable.

```
assignment ::=
        variable := expression ;
        | sliced_variable := expression ;
```

The expression may evaluate to a character string or an aggregate. Each assignment constitutes a redefinition of the variable. Thus a variable with a string value may be reassigned an aggregate value and vice versa.

For an assignment of an aggregate to a sliced variable, each component of the aggregate is assigned to the matching component of the slice. The length of the lefthand slice must be the same as the length of the righthand aggregate. If the slice is of length one, then the righthand side of the assignment may be either a single character string or an aggregate of length one.

Examples:

```
%Status := 1;
%Status := Ada(Prog.VM370.2);
%Plist(1) := (Ada(Prog.VM370));
%alist(2..3) := (green,blue);
```

5.3 PIPELINE STATEMENT

A pipeline statement is used to invoke programs as processes.

```
pipeline_statement ::= pipeline ; | ( pipeline ) ;
```

Unparenthesized invocation of a pipeline causes the ACLI to wait for completion of the execution of the pipeline. A parenthesized pipeline will

61

be executed asynchronously (i.e., the ACLI does not wait for the pipeline to complete. The value returned by a parenthesized pipeline is an aggregate containing the process_ids of each process in the pipeline. The value returned from a pipeline executed as a statement will be discarded.

Examples:

```
list(Radar/Track) | sort();
(list(Radar/Track) *>> saved_errors | sort() -> slist);
(Ada(Prog.OS32.17));
```

5.4 IF STATEMENT

An if statement selects for interpretation one or none of a number of sequences of statements, depending on the truth value of one or more corresponding conditions.

```
if_statement ::=
        if expression then
            sequence_of_statements
        { elsif expression then
            sequence_of_statements }
        [ else
            sequence_of_statements ]
        end if ;
```

For the interpretation of an if statement, the expression after the "if" and the expressions after each "elsif" are evaluated in succession until one of them evaluates to TRUE (see Paragraph 3.3.1). The corresponding sequence of statements is then interpreted. If none of the expressions evaluates to TRUE, the sequence of statements following the "else" is interpreted. If none of the expressions evaluates to TRUE and the "else" is absent, then none of the sequences of statements are interpreted.

Examples:

```
if Ada(Prog) /= ok then
     echo("Unsuccessful execution");

if %alist(2) = green then
     Ada(Prog);
elsif %alist(2) = yellow then
     Link(Prog);
else
     Prog(%alist);
```

5.5 CASE STATEMENT

A case statement selects for interpretation one of a number of alternative sequences of statements, depending on the value of an expression.

```
case_statement  ::=
     case expression is
        { when choice { | choice } => sequence_of_statements }
     end case ;
choice  ::=  expression  |  others
```

Each alternative sequence of statements is preceded by a list of choices specifying the values for which the alternative is to be selected. The expression and each of the choices must evaluate to single character strings (i.e, not aggregates). The choice others may only be given as the choice for the last alternative, to cover all values not given in previous choices.

The values of the choices need not be determinable statically. Once the expression is evaluated, the choices are evaluated in order until one of them matches the value of the expression. The remaining choices are not evaluated.

Example:

```
%stat := Ada(prog);
case %stat is
     when ok =>
          echo("no errors");
          Link(Prog);
     when warning =>
          echo("warnings");
          Link(Prog);
     when serious | fatal =>
          echo("serious or fatal error, no link");
end case;
```

5.6 LOOP STATEMENT

A loop statement specifies that a sequence of statements is to be interpreted repeatedly zero or more times.

```
loop_statement  ::=
  [ loop_identifier : ] [ iteration_clause ]
          basic_loop [ loop_identifier ] ;
iteration_clause  ::=
     for loop_parameter in aggregate
     |  while expression
basic_loop  ::=
     loop
        sequence_of_statements
     end loop
loop_parameter  ::=  variable
loop_identifier  ::=  identifier
```

The optional loop identifiers appearing at the beginning and end of a loop statement must be identical. If a loop_identifier appears within a loop statement (i.e, in a return or exit statement), then the identifier must appear at the beginning and end of the loop.

A loop statement without an iteration clause specifies repeated interpretation of the basic loop. In this case the loop may be left only by an exit statement, return statement, or goto statement.

The "for" iteration clause indicates that the loop parameter is to be assigned the i^{th} value of the aggregate for the i^{th} iteration through the loop. The loop is interpreted once for each value in the aggregate. If the aggregate is empty the basic loop is not interpreted.

In a loop statement with a "while" iteration clauses, the expression is evaluated and tested before each interpretation of the basic loop. If the expression evaluates to TRUE the basic loop is interpreted, if FALSE the interpretation of the loop is terminated.

Examples:

```
for %p in # list(Radar) #
     loop
          Ada(%p);
     end loop;

%i := 1;
while Ada(%plist(i)) = ok
     loop
          link(%plist(i));
          %i := %i + 1;
     end loop;
```

5.7 EXIT STATEMENT

An exit statement may cause the termination of a loop, depending on the truth value of an expression.

```
exit_statement  ::=
      exit [ loop_identifier ] [ when expression ] ;
```

The loop exited in the innermost loop unless the exit statement names the loop_identifier of an enclosing loop, in which case the named loop is exited (along with any enclosing loop inner to the named loop.

If an exit statement contains a when clause, the expression is evaluated and loop termination occurs if and only if the truth value of the expression is TRUE.

An exit statement may only appear within a loop, and a named exit statement only within the named loop.

Example:

```
cloop: for %p in # list(Radar) #
          loop
               exit cloop when Ada(%p) /= ok;
          end loop cloop;
```

5.8 RETURN STATEMENT

A return statement terminates the interpretation of an ACL function or procedure.

```
return_statement  ::=  return [ expression ] ;
```

A return statement may only appear within a function or procedure body. A return statement for a procedure must not include an expression.

Examples:

```
return 3 ;
return green ;
return Ada(%p) ;
```

5.9 INLINE STATEMENT

An inline statement provides for inline expansion of command language procedures.

```
inline_statement  ::=
       . mcl_program_name ( [ actual_parameter_part ] ) ;
```

The named program must be an ACL procedure. The procedure body is expanded inline and is thus interpreted in the context of the present invocation of the ACLI. This is to be contrasted with the call described in Paragraph 4.1, which required that a separate ACLI process be created to interpret the procedure.

Example:

```
. chgwpar("/Radar/Track/Sub2");     -- chgwpar is an ACL subprogram that
                                    -- assigns its argument to %CWP, this
                                    -- assignment would take effect in the
                                    -- current instance of the ACLI
```

5.10 RENAMING STATEMENT

A renaming statement is used to provide an alias for an identifier appearing as a program name.

renames_statement ::= identifier renames identifier ;

The identifier on the left will be considered to be an alias for the identifier on the right, but only when the identifier serves as a program name (see 4.1). The alias is valid when it is the value of a variable used as a program name.

Examples:

```
adacompiler renames Ada;
l renames list;
s renames sort;
```

SECTION 6 - COMMAND LANGUAGE SUBPROGRAMS

An ACL subprogram defines a procedure or function to be interpreted by the ACLI. Exactly one subprogram may appear in a data base object (file). Invocation of the subprogram is indicated by naming the data base object (see Paragraph 4.1).

6.1 SUBPROGRAM DECLARATIONS

A subprogram declaration defines a procedure or function.

```
subprogram_declaration ::=
     subprogram_specification is subprogram_body
subprogram_specification ::=
     procedure [ formal_part ]
     | function [ formal_part ]
formal_part ::=
     ( parameter_declaration { ; parameter_declaration } )
parameter_declaration ::=
     variable { , variable } mode [ := expression ]
mode ::= [ in ] | out | in out
subprogram_body ::= sequence_of_statements
```

As indicated above, the name of the procedure or function is the name of the containing data base object. Interpretation of a subprogram involves parameter association (see Paragraph 6.2) and interpretation of the subprogram body.

The subprogram specification must not contain any substitutes (see Paragraph 4.3).

6.2 PARAMETER ASSOCIATION

Actual parameters may be passed in positional order (positional parameters) or by explicitly naming the corresponding formal parameters (named parameters). For positional parameters, the actual parameter corresponds to the formal parameter with the same position in the formal parameter list. For named parameters, the corresponding formal parameter is explicitly given in the call. Named parameters may be given in any order.

Positional parameters and named parameters may be used in the same call provided that positional parameters occur first at their normal position. That is, once a named parameter is used, the rest of the call must use only named parameters.

"In" and "in out" parameter declarations may provide a default expression. This expression is to be assigned to the specified formal parameter if and only if a corresponding actual parameter does not appear in the call. Actual parameters may be omitted from a call if and only if the corresponding formal parameter declaration provides a default expression. In such a case the rest of the call, following any initial positional parameters, must use only named parameters.

All parameters behave as local variables within the subprogram. When a "return", or the last statement in the subprogram, is interpreted, the values of all "out" and "in out" parameters are copied to their corresponding actual parameters.

By convention, the value returned from a function invocation will usually be interpreted as a "return" or "status" code. The ACLI will provide a return code of "0" for procedures that terminate normally, and a "1" for procedures that terminate abnormally and for functions that terminate before executing a return statement.

Example:

The following is a subprogram, copar, to compile and link all objects within a partition. It can be invoked by "copar(partition_name)".

```
function(%partition : in) is
     for %i in # list(%partition) #
          loop
               if ada(%i) /= ok then
                    return "bad compile" & %i;
          end loop;
     %s := link(# list(%partition) #, PAROUT);)
     if %s = ok then
          return success;
     else
          return "link failure";
```

ANNEX A - STANDARD ENVIRONMENT

Each process has a Standard Environment (including, for example, definitions for Standard Input, Standard Output, Standard Error), and the ACLI is no exception. Certain components of the environment are directly accessible in ACL through preset variables. The ACL user is free to alter the values of these variables at any time. The following environment variables are defined:

- %PATH - A list of partitions, delimited by colons (":"), that are applied in order in searching for an object that is specified with an incomplete name. PATH is initialized to ":/mapse/bin", indicating that object names are first looked for in the current partition, and then in "/mapse/bin".
- %CWP - The full path name of the current working partition. CWP is initialized to the user's "home" partition, which is found in the Password Object [].
- %HOME - The "home" partition for the user, initialized to the value found in the Password Object.
- %TERM - The type of terminal for which output is to be prepared. Of necessity, the permissable values for TERM are dependent on the host environment.
- %CURBRANCH - The string to be used as the branch qualifier when an unqualified name is specified for an abstract object.

The items listed above will be included in the Standard Environment of any process created by the ACLI. There is an additional ACL variable that is strictly local to a ACLI invocation:

%PROMPT -- The string that the ACLI will use to prompt a user on an interactive terminal.

ANNEX B - PROFILES

In each user's home directory there may exist an ACL object with the name "PROFILE". When initially attached to an interactive terminal, the ACLI will, if the "PROFILE" object exists, read and interpret the ACL statements in that object before accepting input from the terminal.

This facility enables the user to alter the default Standard Environment by setting the variables %PATH, %CWP, %HOME, %TERM, %CURBRANCH, and %PROMPT. The user will often include a list of renaming statements to allow shortened aliases of frequently invoked programs. While the foregoing is the typical use for profiles, the PROFILE object may include any ACL statements.

13

ANNEX C - COMMAND UTILITIES

While technically not part of the ACL syntax, there are a number of utility programs that are essential to providing minimal command language functionality. The Command Utilities are logically separate programs. They may be written either in Ada or in ACL. Those Utilities that affect the execution state of the ACLI will be implemented within the ACLI as subprograms. All other Utilities may appear in command pipelines and are invoked as separate processes.

The Command Utilities make available to the ACLI user the applicable functionality provided by the KFW Interface Package and the KDBS Utility Package. Of particular interest are the Find Object and Read Attribute Value Utilities, which can be combined to generate a variiety of data base reports.

Below is a list of the Command Utilities to be provided initially, individual descriptions are on the following pages. Just as the Compiler, Editor, Linker, Loader, and ACLI constitute only an initial set of System Tools, so the following constitute only an initial set of Command Utilities. All Command Utilities read from the Standard Input File, write output to the Standard Output File, and write diagnostics to the Standard Error File.

COMMAND UTILITIES

Add Attribute-Value	- adda	Find Objects	- fobj
Add Value	- addv	Group	- group
Archive	- archive	Help	- help
Branch Create Access	- bca	Link	- link
Branch Write Access	- bwa	List Access Rights	- lacc
Change Value	- chgv	List Attributes	- lattr
Copy	- copy	List Groups	- lgrp
Create	- create	List Partition	- list
Create Branch	- createb	List Processes	- listproc
Create Group	- cgrp	List Versions	- listv
Create Partition	- cp	Load Program	- loadp
Date/Time	- date	Print	- print
Debug	- debug	Read Access Rights	- racc
Delete Attribute	- dela	Release Process	- relp
Delete Group	- dgrp	Read Attribute Value	- rdav
Delete Object	- dl	Set Access Control	- sacc
Delete Partition	- dlp	Set Default Branch	- sdefb
Delete Value	- delval	Suspend Process	- susp
Echo	- echo	Terminate Process	- termp

Add Attribute-Value

This function adds an attribute with an initial value to an object in the KAPSE data base. The user must have mod access to the object.

Calling Syntax:

```
adda ( object_name, attribute_name, attribute_value );
```

Arguments:

object_name	-	The name of the object in which the attribute-value pairs is to be added.
attribute_name	-	The name of the attribute to be associated with the object.
attribute_value	-	The initial value to be assigned to the specified attribute.

Example:

```
adda ( TARGET_PGM, SYSTEM, TANK_SYSTEM );
```

Add Value

This function adds another value to an existing attribute of an object. The user must have mod access to the object.

Calling Syntax:

```
addv (object_name, attribute_name, attribute_value );
```

Arguments:

object_name	-	The name of the object to which the attribute value is to be added.
attribute_name	-	The name of the attribute to which the value is to be added.
attribute_value	-	The value to be added to an existing attribute.

Example:

```
addv ( RADAR_PGM, CONF_LIST, SIGNAL_GEN );
```

Archive

This function manipulates archive objects in the KAPSE data base. Archive handles creating and listing archive objects, and adding, replacing, updating, and deleting the members of an archive object. An archive object is created when an object is to be added or replaced and the specified archive object does not exist.

Calling Syntax:

```
archive ( archive_function, archive_object_name, object_name );
```

Arguments:

archive_function - Indicates the type of archive function to perform:

append - The specified object is to be added to the archive object.

delete - The specified object is to be deleted from the archive object.

list - The membership list is to be printed on Standard Output.

replace - The specified object is to replace the member of the same name in the archive object.

update - The specified object is to replace the member of the same name in an archive only when the specified object is more recent than the corresponding member.

archive_object_name - The name of the archive object. If it does not exist in processing an append or replace operation, the archive object is created.

object_name - The name of the object for which the archive function is to be performed.

Examples:

```
archive ( append, /USAF_PROJ/F16_ARCHIVE, F16_PGM );

archive ( list, /USAF_PROJ );
```

Branch Create Access

This function adds or deletes the right to add a branch to an abstract object for a user or user group.

Calling Syntax:

```
bca ( object_name, user_name, mode );
bca ( object_name, group_name, mode);
```

Arguments:

object_name	-	The abstract object to which branch create permission is to be added or deleted.
user_name	-	The user for whom branch create permission is to be added or deleted.
group_name	-	The group for which branch create permission is to be added or deleted.
mode	-	"add" to add version permission, "del" to delete version permission.

Examples:

```
bca(TARGET_PGM,fred,del);
bca(TARGET_PGM,TESTTEAM,add);
```

Branch Write Access

This function adds or deletes the right to add a version to a branch of an abstract object for a user or user group.

Calling Syntax:

```
bwa ( object_name, user_name, mode );
bwa ( object_name, group_name, mode);
```

Arguments:

object_branch_name	-	The object branch name to which version permission is to be added or deleted.
user_name	-	The user for whom version permission is to be added or deleted.
group_name	-	The group for which version permission is to be added or deleted.
mode	-	"add" to add version permission, "del" to delete version permission.

Examples:

```
bwa(TARGET_PGM,fred,del);
bwa(TARGET_PGM,TESTTEAM,add);
```

Change Value

This function changes the value of an attribute for a specified object in the KDB. For most system-defined attributes, Change Value may only be executed by the System Administrator.

Calling Syntax:

```
chgv ( object_name, attribute_name, attribute_value );
```

Arguments:

object_name	-	The name of the object for which the attribute value is to be changed.
attribute_name	-	The name of the attribute whose value is to be changed.
attribute_value	-	The new value to be assigned to the attribute of the specified object.

Example:

```
chgv ( TEST_PGM, SYSTEM, PAYROLL);
```

Copy

This function copies the informational contents of one object into another. If the second argument names a partition object, an object of the same name as the first is created in that partition if such an object does not exist.

Calling Syntax:

```
copy ( to_object_name );
copy ( from_object_name, to_object_name );
copy ( from_object_name, to_partition_name );
```

Arguments:

from_object_name	-	The name of the object that is to be copied. If not specified, the standard input file is copied.
to_object_name	-	The name of the object to which the first object is to be copied.
to_partition_name	-	The name of the partition containing the destination object.

Example:

```
copy ( SUB_PGM, /NAVY_PROJ/SUB_SYSTEM );

copy ( NEW_SUB_SYSTEM );
```

Create

The create function creates an object in a specified partition.

Calling Syntax:

```
create ( object_name );

create ( object_name, version_type );

create ( object_name, KEYED );

create ( object_name, KEYED, version_type);
```

Arguments:

object_name	The name of the object to be created. If keyed is specified, the object will have keyed access.
version_type	the type of version control to be provided for the object. If no value is specified, the object will not be subject to version control. The types of version control are:

copy - Indicates that copies of the versions are to be kept.

delta - Indicates that delta versioning is in effect for the specified object.

Example:

```
create ( /NAVY_PROJ/SUB_SYSTEM/SONAR_PGM, delta );

create ( TEST_PGM );
```

Create Branch

This function adds a branch to an abstract object.

Calling Syntax:

createb (object_version_name, branch_name);

Arguments:

object_version_name	-	The version at which a branch is to be sprouted.
branch_name	-	The name of the branch to be sprouted.

Example:

createb (TEST_PGM.DEV.3, REL);

Create Group

This function creates a new user group.

Calling Syntax:

```
cgrp ( group_name );
```

Arguments:

group_name - The name of the new group.

Example:

```
cgrp ( TEST_TEAM );
```

Create Partition

This function creates a KAPSE data base partition.

Calling Syntax:

```
cp ( partition_name );
```

Arguments:

partition_name	-	The name of the partition to be created.

Example:

```
cp ( /ARMY_PROJ/TANK_SYSTEM );
```

Date/Time

This function retrieves the current date and time. The value returned from the function is of the form: "mm/dd/yy hh:mm:ss".

Calling Syntax:

```
date();
```

Example:

```
%a := date();
```

Debug

This is an internal ACL call that transfers control to the Debugger.

Calling Syntax:

```
debug(process_id);
debug();
```

Arguments:

process_id	-	Identifies the process to be debugged. If absent, the Debugger will request further information.

Examples:

```
debug();
debug(p417);
```


9C

Delete Attribute

This function deletes an attribute from an object.

Calling Syntax:

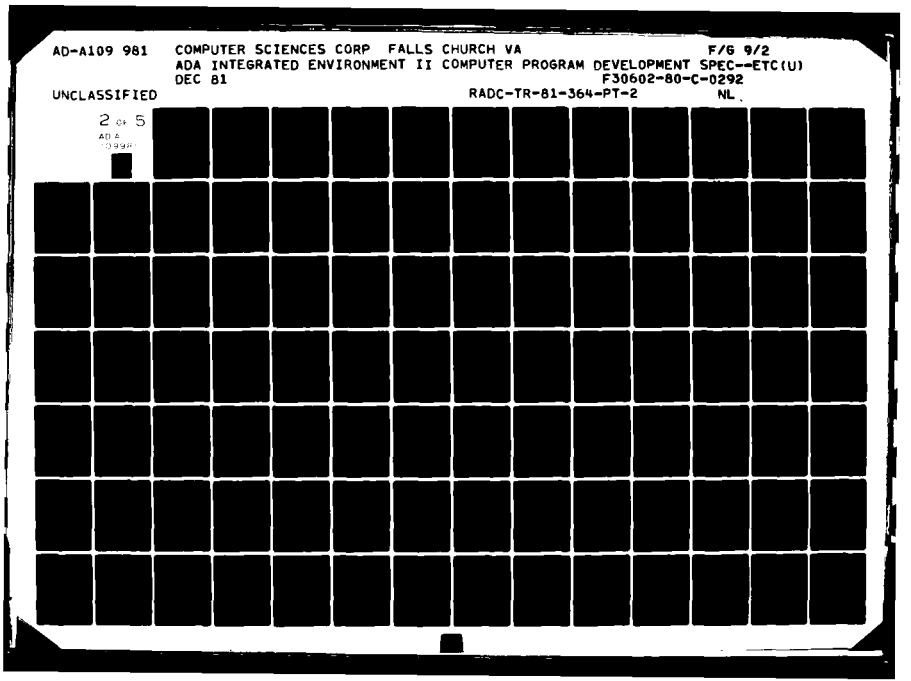
```
dela (object_name, attribute_name );
```

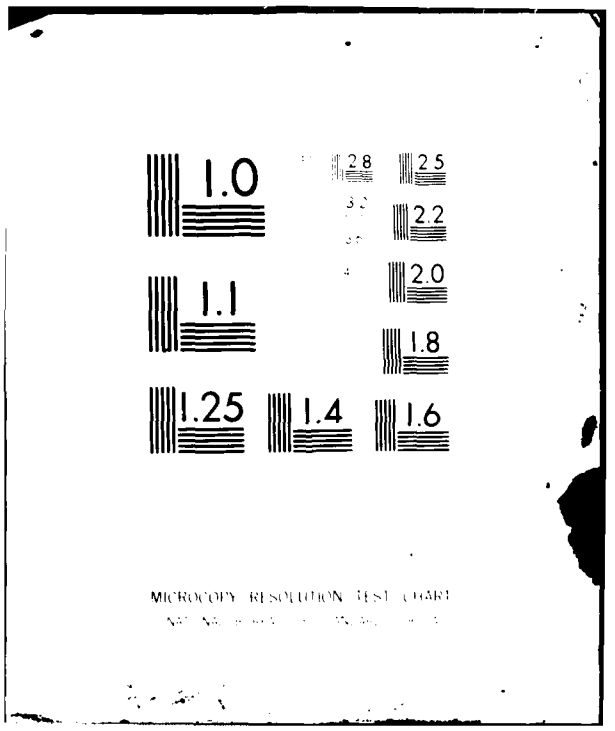
Arguments:

object_name	-	The name of the object whose attribute is to be deleted.
attribute_name	-	The name of the attribute to be deleted.

Examples:

```
dela ( TEST_PGM, SYSTEM );
```


AD-A109 981
COMPUTER SCIENCES CORP FALLS CHURCH VA
F/G 9/2
ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPEC--ETC(U)
DEC 81
F30602-80-C-0292
UNCLASSIFIED
RADC-TR-81-364-PT-2
NL
2 of 5


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
MICROCOPY RESOLUTION TEST CHART

Delete Group

This function deletes a user group.

Calling Syntax:

```
dgrp ( group_name );
```

Arguments:

group_name	-	The name of group to be deleted.

Example:

```
dgrp ( USAF_GROUP );
```

Delete Object

This function deletes an object from the KAPSE data base.

Calling Syntax:

```
dl ( object_name );
```

Arguments:

object_name - The name of the object to be deleted.

Example:

```
dl ( /NAVY_PROJ/SUB_SYSTEM/SUB_PGM );
```


93

Delete Partition

This function deletes a *partition from* the KAPSE data base.

Calling Syntax:

```
dlp ( partition_name );
```

Arguments:

partition_name - The name of the partition to be deleted .

Example:

```
dlp ( /ARMY_PROJ/TANK_SYSTEM );
```

Delete Value

This function deletes a value from a list of values for an attribute. This function is not to be used in deleting attributes with a single value.

Calling Syntax:

```
delval ( object_name, attribute_name attribute_value );
```

Arguments:

object_name	-	The name of the object from which the value of an attribute is to be deleted.
attribute_name	-	The name of the attribute from which a value is to be deleted.
attribute_value	-	The value of the attribute which is to be deleted.

Example:

```
delval ( TEST_PGM, CONF, PAYROLL_PGM );
```

Echo

This function prints its argument.

Calling Syntax:

```
echo ( print_string );
```

Arguments:

print_string - The string to be output to Standard_Ouput.

Examples:

```
echo ( %A );

echo ( abc );

echo ( "PRINT THIS STRING" );
```

Find Objects

For a given partition, this function finds all the objects with attribute values satisfying a specified boolean predicate.

Calling Syntax:

```
fobj ( partition_name, predicate, search );
```

Arguments:

partition_name - The name of the partition to be searched.

predicate - A character string containing a boolean expression with the following syntax:

```
expression ::=
      relation { and relation }
    | relation { or relation }
    | relation { xor relation }
    | ( expression )

relation ::=
      attribute_name = value
    | attribute_name /= value
    | value in attribute_name
    | value not in attribute_name

value ::=
     character_string
```

search - A value of "yes" or "no". This indicates whether subpartitions of the specified partition are to be searched as well.

Example:

```
fobj ( NAVY_PROJ, "system=sub or system=dest", yes );

fobj ( NAVY_PROJ, "co_AIR in cfg_list", no );
```

Group

This function can add to, delete from, or list the membership of a user group.

Calling Syntax:

```
group ( action, group_name, user_name_list );
```

Arguments:

action - The type of function to be performed on the group:

- a - The user names are to be added to the group.
- d - The user names are to be deleted from the group.
- l - The list of members of the group is to be printed.

group_name - The name of the user group.

user_name_list - A character string containing a blank-delimited list of user names.

Example:

Add a user to a defined group:

```
group ( a, TEST_TEAM, DAVE );

group ( d, TEST_TEAM, "Bill Mary" );

group ( l, TEST_TEAM );
```

Help

This function obtains and prints information about the APSE system functions. This information is in the form of an explanation of the usage of the specified function of the APSE.

Calling Syntax:

```
help ( object_base_name );
```

Arguments:

object_base_name	-	An object with a name of the form object_base_name'HLP is searched for and printed

Example:

```
help ( d1 );
```


100

Link

This function creates a link in the current working partition to the named object.

Calling Syntax:

```
link(link_name, object_name);
```

Arguments:

link_name	-	The name of the link.
object_name	-	The full pathname of the object to which the link is created.

Example:

```
link(TST_PGM,/USAF/RADAR/TRACK/TEST/TST_PGM);
```


101

List Access Rights

This function lists the access control attribute for a specified object in the KDB.

Calling Syntax:

```
lacc ( object_name );
```

Arguments:

object_name	-	The name of the object for which the access control attribute is to be listed.

Example:

```
lacc ( TEST/SUB_SYS );
```

List Attributes

This function lists the attribute names of a specified object in the KAPSE data base.

Calling Syntax:

```
lattr ( object_name );
```

Arguments:

object_name	-	The name of the object for which the attribute names are to be listed. Note that the values of the attributes are not listed.

Example:

```
lattr ( /NAVY_PROJ/SUB_SYS/SONAR_PGM );
```

List Groups

This function lists all user groups. The individual members of the groups are not listed. (see the Group function for this facility).

Calling Syntax:

```
lgrp ();
```

Example:

```
lgrp ();
```


IC V

List Partition

This function lists the members of a partition in the KAPSE data base.

Calling Syntax:

```
list ( partition_name );
```

Arguments:

partition_name - The name of the partition to be listed.

Example:

```
list ( /USAF_PROJ/RADAR );
```

List Processes

This facility is provided to list information about all processes that are descendants of the current instance of the ACLI. Information listed includes process id, corresponding Load Object name, and status.

Calling Syntax:

```
listproc ();
```

Arguments:

None.

Example:

```
listproc ();
```

List Versions

This function generates a listing identifying all versions, along with associated information, for an abstract object.

Calling Syntax:

```
listv(object_name);
```

Arguments:

object_name	-	The abstract object for which the version information listing is to be generated.

Example:

```
listv(/radar/track/sys3/testprog);
```

Load Program

This function loads a Load Object as a process in a suspended state. This capability enables the debugger to modify the process before it executes. Loadp returns the process_id of the created process.

Calling Syntax:

```
loadp ( object_name, ( argument_list ) );
```

Arguments:

object_name	-	The name of the Load Object to be loaded as a suspended process.
argument_list	-	The arguments to be passed to the process when it is released.

Example:

```
%pid = loadp ( ada, ( prog'TXT ) );
```

Print

This function prints the informational content of an object in the KAPSE data base.

Calling Syntax:

```
print ( object_name );
```

Arguments:

object_name	-	The name of the object whose informational content is to be printed.

Example:

```
print ( /SUB_SYS/TRAINER_PGM );
```

Read Access Rights

This function lists the access rights of a specified user or group for an object in the KDB.

Calling Syntax:

racc (object_name, name);

Arguments:

object_name	-	The name of the object from which the access rights are to be retrieved.
name	-	A user_name or group_name for which the access rights are read.

Examples:

racc (TEST_PGM, TEST_TEAM);

Read Attribute Value

This function lists the value of an attribute for an object in the KAPSE data base.

Calling Syntax:

```
rdav ( object_name, attribute_name );
```

Arguments:

object_name	-	The name of the object from which the value of an attribute is to be retrieved.
attribute_name	-	The name of the attribute whose value is to be listed.

Examples:

```
rdav ( TEST_PGM, SYSTEM );
```

Release Process

This function releases a specified MAPSE process that has previously been suspended. Only an invocation of the ACLI that is an ancestor of the process is permitted to release it.

Calling Syntax:

```
relp ( process_id );
```

Arguments:

process_id - The id of the process that is to be released.

Example:

```
relp ( p315 );
```

Set Access Control

This function sets the access control rights for a particular object in the KAPSE data base.

Calling Syntax:

```
sacc ( object_name, user_name, access_rights );
```

Arguments:

object_name - Name of the object to which the access control rights are to be assigned.

user_name - The name of a particular user or group for which access rights are to set. The reserved name "default" indicates that rights are to be set for "all other" users.

access_rights - The assigned access rights can be a combination of any of the following:

r - Read permission.

w - Write permission.

e - Execute permission.

d - Delete permission.

a - Append permission.

m - Access control modification permission.

p - Indicates that the access control string is for partition entries rather than for the partition object.

The - and the + have special meaning in defining the access control rights for an object. The + is used to add additional access rights for a particular user, while - is used to delete particular access rights. A string with - alone indicates that a particular user may be assigned to have no access rights to the specified object. If the access_rights string is prefixed with a p, the rights are assigned to the partition access attribute for the object_name, which must name a partition.

Examples:

Define access to an object:

```
sacc ( /NAVY_PROJ/RADAR/TRACK, rwe );
```

Add modify permission:

```
sacc ( /USAF_PROJ/RADAR/TRACK, +m );
```

Delete execute permission:

```
sacc ( /USAF_PROJ/RADAR/TRACK, -e );
```

Deny access to a particular user:

```
sacc ( /USAF_PROJ/RADAR/TRACK, - );
```

Set partition access:

```
sacc ( /USAF_PROJ/RADAR, TEST_TEAM, prwe );
```

Set Default Branch

This function identifies, for an abstract object, the branch that is to be used as the default (or preferred) branch. On a default branch, the last version will always be the default version.

Calling Syntax:

```
sdefb(object_branch_name);
```

Arguments:

object_branch_name	-	The name of the branch to be used as the default

Example:

```
sdefb(/radar/track/prog.rel1);
```

Suspend Process

This function suspends all tasks within a specified MAPSE process. Only an invocation of the ACL1 that is an ancestor of the process is permitted to suspend it.

Calling Syntax:

```
susp ( process_id );
```

Arguments:

process_id	-	The id of the process that is to be suspended.

Example:

```
susp ( p315 );
```

Terminate Process

This function terminates a MAPSE process. Only an invocation of the ACLI that is an ancestor of the process is permitted to terminate the process.

Calling Syntax:

```
termp ( process_id );
```

Arguments:

process_id	-	The id of the process that is to be terminated.

Example:

```
termp ( p315 );
```

Volume 4

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

Configuration Management System

Prepared for

Rome Air Development Center
Griffiss Air Force Base, NY 13441

Contract No. F30602-80-C-0292

119

TABLE OF CONTENTS

120

TABLE OF CONTENTS (Cont`d)

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This document presents the Computer Program Development Specification (Type B5) for the Computer Program Configuration Item (CPCI) known as the Minimal Ada Programming Support Environment (MAPSE) Configuration Management System (CMS). This specification establishes the performance, design, and test requirements for the CMS.

1.2 FUNCTIONAL SUMMARY

The Configuration Management System consists of a MAPSE tool, Configure, that interacts with the KAPSE Data Base System (KDBS) to process and maintain configurations and object histories. A configuration is a set of data base objects combined with a set of rules that specify how these objects may be derived from each other.

SECTION 2 - APPLICABLE DOCUMENTS

2.1 PROGRAM DEFINITION DOCUMENTS

1. Requirements for Ada Programming Support Environment, STONEMAN, February 1980.

2. Statement of Work, Contract No. F30602-80-C-0292, 26 March 1980.

2.2 INTER-SUBSYSTEM SPECIFICATIONS

3. System Specification for the Ada Integrated Environment.

4. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.

5. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.

6. Volume 3, Computer Program Development Specification for CPCI KAPSE APSE Command Language Interpreter.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

7. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.

8. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

9. Aho, A. V., and J. D. Ullman, The Theory of Parsing, Translation, and Compiling, Vols. I and II, Prentice-Hall, 1972.

10. Aho, A.V., J. E. Hopcroft, and J. D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section provides the general description, identifies the external and internal interfaces, and presents the functional requirements and internal characteristics of the Configuration Manager.

3.1.1 General Description

The Configuration Manager processes configurations and object histories. It is a MAPSE-level tool, Configure, that utilizes KDBS functions to manage the history attributes.

A configuration is defined in a configuration object. The two major functions of Configure are:

Update - Determine and perform the minimal set of operations that are required to bring specified objects in a configuration up-to-date with the other objects in the configuration.

Reconstruct - Determine and perform the minimal set of operations that are required to reconstruct a specified version of an object in a configuration.

3.1.2 Peripheral Equipment Identification

Configure is a host-independent tool. The KAPSE data base objects processed by Configure reside on secondary storage, typically host disk files.

3.1.3 Interface Identification

Configure invokes the APSE Command Language Interpreter (ACLI) [6] to process derivation specifications. Configure is the sole user of those KDBS functions that manage the history attributes [5]. In addition, Configure interfaces with the Ada Standard I/O Package, provided by the KAPSE Data Base System (KDBS), and the KAPSE Framework (KFW) Interface Package, provided by the KFW [4].

3.2 FUNCTIONAL DESCRIPTION

This paragraph describes the functions of the Configuration Manager, the program and equipment relationships and interfaces identified above, and the input/output utilization.

3.2.1 Equipment Description

Configure is designed to be highly portable, and will execute on both the IBM VM/370 and the Interdata 8/32. All interfaces to the KAPSE are through the KAPSE virtual interface, and Configure is completely host-independent.

3.2.2 Computer Input/Output Utilization

Configure processes KAPSE data base objects that will typically be represented as files on host storage devices.

3.2.3 Computer Interface Block Diagram

See Figure 3-1.

3.2.4 Program Interfaces

This Paragraph identifies the interfaces between Configure and the other MAPSE components. Specifically, Configure has interfaces to the ACLI and to the history management functions of the KDBS. Configure also interfaces with the Ada Standard I/O Package provided by the KDBS, and with the KFW Interface Package to invoke the ACLI.

3.2.4.1 The ACLI Interface

A configuration object contains prototype commands written in the APSE Command Language (ACL). Configure performs some initial processing of these commands, and then creates a separate ACLI process to execute them. The ACLI process is invoked using the "Start_Process" function available in the KFW Interface Package. The ACLI process returns a status code indicating the successful execution of the command or the reason why the execution failed.

3.2.4.2 The KDBS Interface

Configure calls on KDBS functions to create and to modify the history attributes of objects. The detailed specification of these functions are

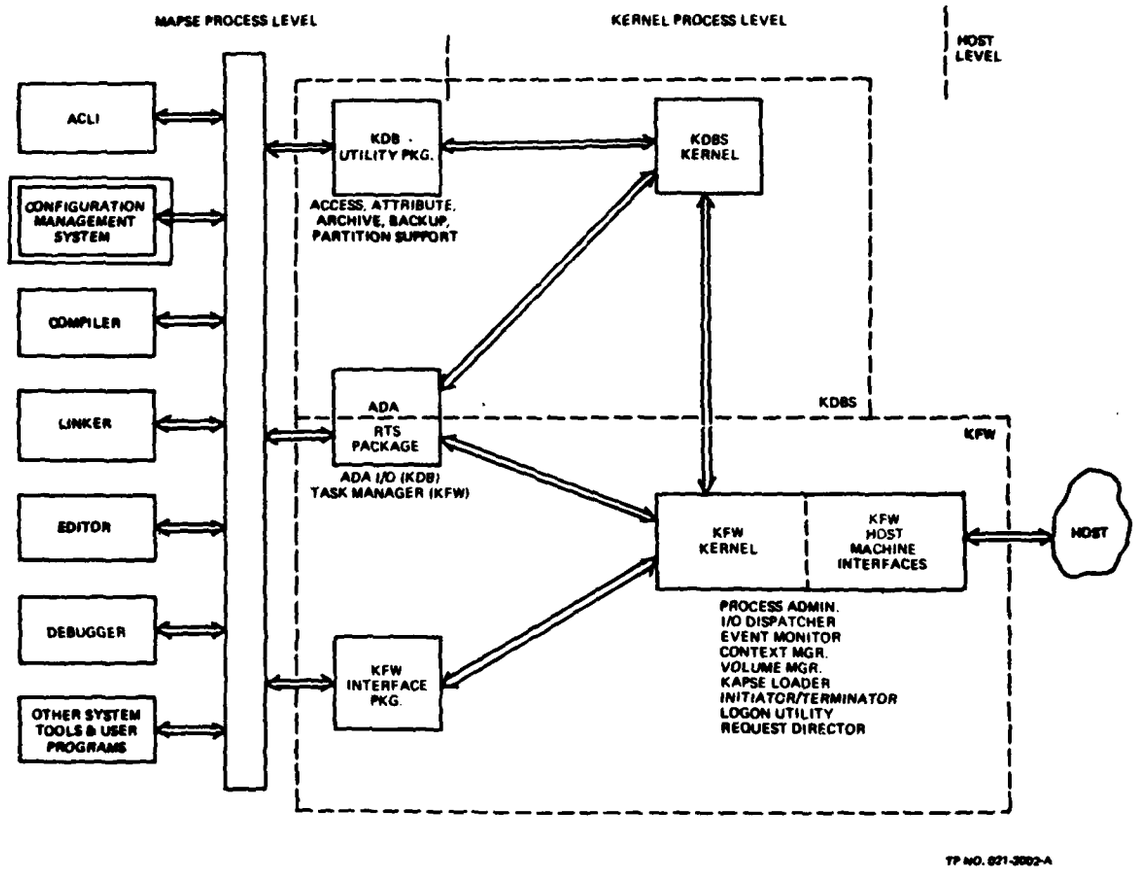


Figure 3-1. Interface Diagram

126

provided in [5]. The history attributes consist of four individual attributes: date_time, ref_count, cfg_list, and dep_list. The semantics of these attributes are provided in Paragraph 3.2.5. The KDBS utility functions that manipulate them will be invoked as follows:

To read an attribute --

readv("object-name","attribute-name",attribute-value)

To add a new attribute-value --

addav("object-name","attribute-name","initial-value")

To add a value to an attribute list --

addv("object-name","attribute-name","value");

To change the value of an attribute --

chgv("object-name","attribute-name","value");

3.2.5 Function Description

The Configuration Manager is provided to maintain and process configurations, and to permit the reconstruction and maintenance of member objects. In order to present the functionality of Configure, the underlying concepts must first be defined.

A configuration object (CO) defines a set of related objects along with a set of operations to be performed on them. A CO is a text object, and may be created with the MAPSE Editor. COs are created with the category "cfg".

Each CO identifies a set of derived objects. A derived object is one that is produced by applying MAPSE programs to other derived objects and/or to base objects. Within a CO, all objects whose derivations are not specified by that CO are considered to be base objects. A base object in one CO may be a derived object in another CO.

It is possible for a CO to specify no derived objects. In this case, the purpose of the CO would be to define common operations to be performed on the specified base objects - operations that result in no new data base objects.

The operations defined within a CO are written as sets of prototype commands in the APSE Command Language (MCL). Each command set is associated with a

dependency list. A dependency list indicates that a group of derived objects is to be constructed (or configured) by processing a second group of objects. This second group may contain derived objects that are, in turn, dependent on other objects. The collection of dependency lists defines an acyclic directed dependency graph. Each node is linked to every node that it immediately depends on.

The arguments supplied to an invocation of Configure specify either a list of derived objects that are to be brought up to date relative to the objects that they depend on, or a list of object versions whose information content is to be reconstructed. An object is up-to-date when it is at least as new as the objects it depends on; note that this definition must be interpreted rather carefully with respect to versioned objects, Paragraph 3.3.4.2. The primary function of Configure is to perform the minimal processing necessary to update or to reconstruct the specified objects.

One side-effect of updating a configuration is the setting of the history attributes of the affected objects. The history attributes are maintained to guarantee the reconstructability of configured objects. An object has four history attributes:

1. Date-time attribute - identifies the date and time that the object was last modified.
2. Dep-list attribute - identifies the objects (and versions if these are abstract objects) that this object immediatedly depends upon.
3. Ref-count - indicates the number of objects constructed with Configure that immediately depend on this object. Versions of base objects may not be deleted if their ref-count is non-zero.
4. Cfg-list - lists the names of the Configuration Objects that reference this object. (This attribute is optional. If present it allows the determination of all configurations that include a given object.)

The operation of Configure can now be divided into four major steps:

1. Lexical analysis of the CO (Scanner).
2. Parsing of the CO (Parser).
3. Construction of the dependency graph (Constructor).
4. Determination of the objects that need to be updated or reconstructed and invocation of the ACLI to process the associated commands (Updater).

Figure 3-2 illustrates the flow of control between these steps.

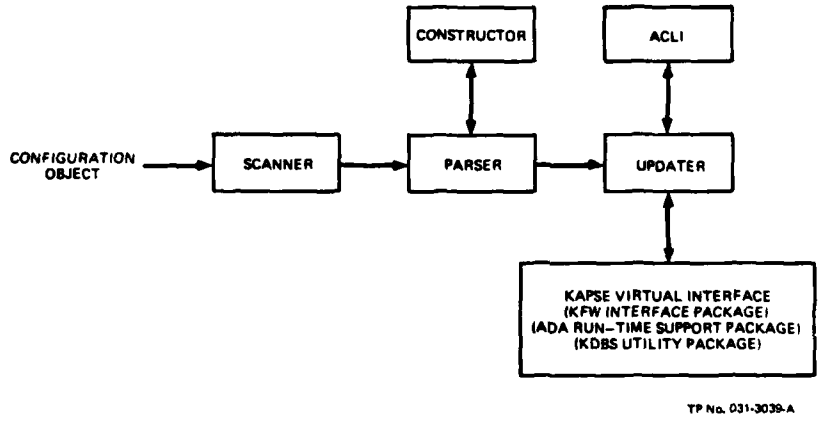


Figure 3-2. Control Flow

3.3 DETAILED FUNCTIONAL REQUIREMENTS

Configure has two functions: updating objects in a configuration, and reconstructing specific versions of objects in a configuration. These functions involve the processing of configuration objects, the derivation of objects defined by the COs, and the management of history attributes so that the defined objects can be reconstructed at any time. The following paragraphs provided a detailed specification of the format of the CO and of its processing by Configure.

Referring to Figure 3-2, there are four processing steps. These are assigned to the Scanner, Parser, Constructor, and Updater. Each of these steps will be treated is a separate paragraph below.

3.3.1 Scanner

The function of the Scanner is to read the CO, divide the text into lexical tokens, and process the macro definitions. The Scanner takes its input directly from the CO, and outputs the resultant tokens to the Parser.

3.3.1.1 Inputs

There are two classes of inputs to be considered. First, there are the arguments passed to an invocation of Configure. Second, there is the format of the CO itself.

3.3.1.1.1 Invocation of Configure

Configure is invoked with the name of a CO, and a list of names of objects to be updated. Invocation is through the Start_Process function supplied by the KFW Interface Package. Users executing the ACLI invoke Configure with an ACLI command:

Configure(CO-name,object-list);

The object-list is a list of data base object names. These may name abstract objects, object branches, or specific versions (see [5] for a discussion of version control and object naming). The object list need not be specified - the default value is the name of the first object specified in the first dependency rule in the CO. The names in the object list must be defined by the CO.

The Scanner opens and reads from the designated CO. The scanner does not further process the object list argument, this argument is preserved for use by the updater.

3.3.1.1.2 CO Format

The CO is an object created using the Ada Text I/O Package, usually by the Editor. The CO is assigned the category cfg. The format of the CO is as follows:

macro-definitions
%%
dependency-rules

Each macro definition must appear on a separate line, and has the form of a simple ACL assignment statement:

%identifier := character_string

ACL syntax is used for the identifier and character_string (see the ACL Reference Manual, Attachment to the B5 Specification for the ACLI). A macro definition indicates that subsequent occurrences of the %identifier are to be replaced by the character_string. These occurrences may appear in later macro definitions.

A dependency-rule has the following form:

target-object-list : dependent-object-list
    prototype-ACL-commands

Each line that does not begin with a space character (or horizontal tab) begins a new dependency rule. The object lists are blank-delimited lists of object names. An object name may identify an abstract object, object branch, or a specific version. The target objects are those objects that are to be updated if they are not at least as new as the dependent objects. The associated prototype-ACL-commands will be executed by the ACLI, after macro substitution is performed, if any target object needs to be updated.

While these constitute the basic requirements for CO syntax, there are a number of extensions that may be considered for inclusion in the C-level

131

specification. Such extensions include default dependency rules (e.g, a `REL object may implicitly depend on compiling a `TXT object of the same name), and special macros that expand to target or dependency lists to ease the specification of the ACL command strings.

3.3.1.2 Processing

Lexical analysis is performed by a simple, table-driven, finite-state recognizer. Algorithms for such recognizers are well-known and simple to implement (e.g., [9]). Macro definitions are saved and matched against subsequent input before this input is divided into lexical tokens. The prototype ACL commands are not divided into tokens, although they are subject to macro substitution. Each such command is considered to be a single token.

3.3.1.3 Outputs

The output of the Scanner is a stream of (token-number, value) pairs. The token number is of enumeration type, and identifies the class of the token. The value is the character string that was the source representation of the token.

Since the Scanner and parser operate logically as coroutines, each is to be defined as an Ada task. A producer task, the Scanner, generates token pairs for a consumer task, the Parser.

3.3.2 Parser

The Parser serves to group together the tokens provided by the scanner, and to call the appropriate constructor procedures to build the dependency graph.

3.3.2.1 Input

The input to the Parser is the output of the Scanner, a list of token names and their values. As indicated above, the Scanner and Parser perform as a pair of producer and consumer tasks, respectively.

3.3.2.2 Processing

Because the format of the CO is so simple, parsing can be accomplished with an algorithm tailored specifically for the purpose.

The Parser calls a Constructor function to create a dependency graph node for each object, whether a target or dependency object:

create_node(object-name);

The function returns a node index (or pointer). New nodes will not be created for object names that have previously been processed.

For each dependency indicated by a dependency rule, a Constructor function is called to link the two nodes:

link(target-node-index, dependency-node-index);

Link returns a link index (or pointer).

A command node is created for each ACL command string:

create_com_node(ACL-command-string);

This node is associated with the each corresponding dependency link:

add_command(com-node-index, link);

3.3.2.3 Output

There is no explicit output of the Parser. The dependency graph is produced as a side-effect so that it can be then processed by the Updater.

3.3.3 Constructor

The Constructor provides all the subprograms necessary to build the dependency graph. The subprograms are invoked by the Parser.

3.3.3.1 Inputs

The following interfaces are provided by the Constructor to the Parser:

create_node(object-name) returns node-index

link(target-node-index, dependency-node-index) returns link-index

create_com_node(ACL-command-string) returns com-node-index

add_command(com-node-index, link);

3.3.3.2. Processing

The above subprograms are called by the Parser, as indicated in Paragraph 3.3.2.2, to build the dependency graph. The processing associated with each of the subprograms is straightforward. Create_node allocates a new graph node. Link creates a dependency between two graph nodes. Create_com_node creates a command node. Add_command links a dependency to a command node.

3.3.3.3. Outputs

The explicit outputs of the Constructor functions have been described above in Paragraphs 3.3.2.2 and 3.3.3.1. The implicit output is the dependency graph.

3.3.4 Updater

The function of the Updater is to determine the minimal set of operations that are required to update or to reconstruct the specified target objects. This determination is made recursively, following object dependencies and the date-time attribute, and the ACLI is invoked to perform the operations.

3.3.4.1 Inputs

There are two inputs to the Updater. The first is the list of target objects specified in the original invocation of Configure. This list is examined by the Updater to determine the initial set of objects to update. If no list is specified, the default action is to update the first object mentioned in a dependency rule.

The second input to the Updater is the dependency graph structure built by the Parser and the Constructor. This structure identifies all dependencies between objects as well as the ACL command strings that must be executed to fulfill the dependencies.

3.3.4.2 Processing

The Updater provides the functions required for normal updating as well as for version reconstruction. While these functions require substantially similar graph-walking processing, they are sufficiently dissimilar to require separate paragraphs below.

3.3.4.2.1 Normal Updating

Normal updating of an object consists of the processing required to bring that object up to date with respect to all other objects in the configuration. The criterion that the Updater uses to determine whether to update an object is called the update criterion.

If a target or dependency object name specifies an abstract object, the update criterion will be applied to the last version on the default branch of that abstract object. If the object name specifies a branch, the update criterion will be applied to the last version on that branch. If the target specifies a version, then Configure is being asked to reconstruct the information content of that version. Reconstruction is treated in Paragraph 3.3.4.2.2. If the object name specifies a nonversioned object (a nonabstract object), the update criterion will be applied to the object itself.

The update criterion is specified below. Note that the use of the term "object" must be interpreted according to the preceding paragraph.

> Update Criterion: A target object must be updated if it is nonexistent, or if it is older than at least one of the objects it depends on.

This criterion must be evaluated for the entire graph rooted by the target object. The following algorithm may be used to process a dependency graph for a given target:

1. Stack the initial target name if it is not a base object and mark its graph node as "stacked".
2. If the stack is empty, terminate the algorithm. Otherwise let TOP be the top name on the stack.

135

3. If TOP is marked "processed", update it if it is older than any of the objects it immediately depends on, pop it from the stack, and go to step 2. This updating involves the following steps:

   a. Invoke the ACLI to process the ACL command string associated with the dependency rule defining the target object being updated.

   b. Set the date-time attribute of the target object to the current date-time.

   c. Set the dep-list attribute of the target object to include the names of each object the target depended on, and the name of the present CO. These must all be fully qualified version names.

   d. Increment the ref-count attribute of each object (version) depended on.

   e. For each target and dependency object (version) that has a cfg-list attribute, add the name of the present CO to the attribute.

4. If TOP is not marked "processed", then mark it "processed" and stack each name that it depends on that is not marked "stacked" or "processed". Mark "stacked" each name that was stacked. Go to step 2.

3.3.4.2.2 Object Reconstruction

The Updater is also used to reconstruct the previously deleted information content of derived object versions. Reconstruction is made possible through the use of the information present in the history attributes. The following algorithm may be used to reconstruct a target object version:

1. Stack the initial target version name (only if the object version is a derived object).

2. If the stack is empty, terminate. Otherwise let TOP be the top name on the stack.

3. If TOP is marked "processed", invoke the ACLI to process the ACL command string associated with the dependency rule defining TOP. However, first instantiate for each abstract object name or branch object name in the string the respective version name that is provided by the dep-list attribute of object TOP. Pop TOP from the stack and go to step 2.

4. If TOP is not marked "processed", mark it "processed" and stack the name of each object version mentioned in its dep-list that is not marked "processed" or "stacked". Do not, however, stack the names of derived versions that have not had their information content deleted. In addition, do not stack names of versions that were defined in a different CO. For these objects, invoke Configure recursively to reconstruct their information content. Mark "stacked" each name that was stacked in this step. Go to step 2.

3.3.4.3 Outputs

There are no explicit outputs of the Updater.

3.4 ADAPTATION

This section describes any adaptation that might be required to rehost the Configuration Manager.

3.4.1 General Environment

There are no aspects of the general environment that impact rehostability.

3.4.2 System Parameters

Depending upon host characteristics, Configure may execute in a fixed amount of main memory or may request more space dynamically. In the first case parameters must be set to limit the characteristics of the Configuration Objects. These parameters include size of the objects, number of dependencies, number of objects referenced, and total length of ACL command strings.

3.4.3 System Capacities

The memory that Configure can request dynamically may be limited. This limitation may result in diagnostic messages issued by Configure when processing large Configuration Objects.

3.5 CAPACITY

Not applicable.

SECTION 4 - QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the Configuration Management System (CMS). The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the Configuration Management System. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government-approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data are processed. For example, a review of hard copy test data might be used to verify that the values of specific parameters are correctly computed.

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

Table 4-1. Test Requirements Matrix

SECTION	TITLE	INSP.	ANAL.	DEMO.	DATA.	SECTION NO.
3.3.1	Scanner		X		X	4.2.1, 4.2.2
3.3.2	Parser		X		X	4.2.1, 4.2.2
3.3.3	Constructor				X	4.2.2
3.3.4	Updater		X		X	4.2.1, 4.2.2

4.1.1 Subprogram Testing

Following unit testing, individual modules of the CMS shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2 Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

CPCI testing shall be performed on all development software of the Configuration Management System. This specification presents the performance criteria which the developed CPCI must satisfy. The correct performance of the CMS will be verified by testing its major functions. Successful completion of the program testing that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3 System Integration Testing

System integration testing involves verification of the integration of the CMS with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the CMS performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details

regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

The Scanner, the Parser, and the graph-walk algorithms of the Updater shall be validated both by analysis of the constituent algorithms and by formal review of test data. The Constructor shall be validated by formal review of test data.

In order to provide for program testing of the Configuration Manager in advance of the completion of the KFW and KDBS, a facility shall be provided to log rather than execute all calls made on KFW and KDBS functions. This logging facility shall be used to generate test output data from the test input scripts. The test input scripts are a subset of those that will constitute the Configure Validation Test Set described below in Section 4.3. Validation shall be by formal review of test data.

The Configure Validation Test Set shall be augmented with tests for validating each interface described in Paragraph 3.2.4. A program shall be written that will automatically execute the augmented test. Validation of the system integration of the Configuration Manager shall be by formal review of test data.

4.2.1 Analysis of Algorithms

The coded scanning and parsing subprograms shall be inspected to ensure that they conform to provably correct algorithms that have appeared in the literature (e.g., [9]). The graph-walk algorithms of the Updater shall be verified via formal mathematical proof techniques (see [10]). The coded Updater subprogram shall be inspected to ensure that it conforms to the proven algorithms.

4.2.2 Review of Test Data

Drivers shall be written to generate input data and to log output data. Test input scripts and expected test output shall be developed by test personnel in accordance with subprogram and program specifications. Testing shall consist of comparing expected output data with test output data. To

minimize the effort required for developing drivers, a bootstrap approach shall be taken. Thus an initial driver for the Scanner shall be provided, the Scanner shall serve as the driver for the Parser, and the Scanner-Parser shall serve as the driver for the Constructor and Updater.

4.5 ACCEPTANCE TESTING

A Configure Validation Test Set shall be prepared, consisting of a comprehensive set of test scripts along with expected output for each script. Scripts shall be provided to generate and process a considerable variety of Configuration Objects. Both the update and reconstruction funcions shall be extensively tested.

A program shall be written that will automatically execute the Configure Validation Test Set and compare the expected output data with the test output data. The output of this program will be a report documenting the successful or unsuccessful execution of each script in the Test Set. Successful execution of the entire Test Set shall constitute successful completion of the acceptance test. Detailed requirements for the Test Set shall be included as part of the Computer Program Test Plans to be approved by the Government.

SECTION 5 - DOCUMENTATION

5.1 GENERAL

The documents that will be produced during the implementation phase in association with the Configuration Manager are:

1. Computer Program Development Specification (Type B5) - Update
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. User's Manual
6. Rehostability Manual
7. MAPSE Tools Reference Handbook
8. Configure Validation Test Set Manual.

5.1.1 Computer Program Development Specification

The final Configuration Manager B5 Specification will be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II.

5.1.2 Computer Program Product Specification

A Type C5 Specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document will be used to specify the design of the Configuration Manager and the development approach implementing the B5 specification. This document will provide the detailed description that will be used as the baseline for any Engineering Change Proposals.

5.1.3 Computer Program Listings

Listings will be delivered that are the result of the final compilation of the accepted Configuration Manager. Each compilation unit listing will contain the corresponding source, cross-reference, and compilation summary. The source listing will contain the source lines from any INCLUDEd source objects.

144

5.1.4 Maintenance Manual

An Configuration Manager Maintenance Manual will be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the Configuration Manager to be easily maintained by personnel other than the developers. The documentation will be structured to relate quickly to program source. The procedures required for debugging and correcting the Configuration Manager, along with debugging aids that have been incorporated as an integral part of the Configuration Manager, will be described and illustrated. Sample scripts for compiling Configuration Manager components, for relinking the Configuration Manager in parts or as a whole, and for installing new releases will be supplied.

5.1.5 User`s Manual

A User`s Manual shall be prepared in accordance with DI-M-30421, and will contain all information necessary for the operation of the Configuration Manager. Because of the virtual user interface persented by the Configuration Manager, a single manual is sufficient for all host computers. Information relevant to specific hosts will be contained in an appendix. A complete list of all Configuration Manager diagnostic messages will be included with supplemental information supplied to assist the user in locating and correcting ACL errors.

5.1.6 Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual will be prepared that describes step-by-step procedures for rehosting the Configuration Manager on a different computer.

5.1.7 MAPSE Tools Reference Handbook

A MAPSE Tools Reference Handbook shall be produced containing syntax diagrams for all command constructs.

5.1.8 Configure Validation Test Set Manual

A manual shall be provided that describes the procedures for running the Configure Validation Test Set. A machine-readable form of the Test Set shall be provided along with listings of each member of the Test Set.

Volume 5

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

Ada Compiler

Prepared for

Rome Air Development Center
Griffiss Air Force Base, NY 13441

Contract No. F30602-80-C-0292

147

TABLE OF CONTENTS

143

LIST OF ILLUSTRATIONS

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This specification establishes the performance, design, development, and test requirements for the Computer Program Configuration Item identified as the MAPSE Tool Set member, the Ada Compiler, referred to hereafter simply as the Compiler.

1.2 FUNCTIONAL SUMMARY

The purpose of this specification is to define the Ada Compiler being designed as part of the Ada Integrated Environment contract for RADC. This document shall serve to communicate the functional design decisions that have been adopted and to provide a basis for the detailed design and implementation phase.

The Ada Compiler is the MAPSE tool which processes Ada source programs and translates these programs into machine level instructions to permit the programs to execute on the selected target computer. The design chosen has been selected to meet the following general requirements and goals:

1. The Compiler must implement the full Ada Language as described in reference 1. This is an absolute requirement.

2. The Compiler must generate very efficient object code. One of the primary uses of the language will be for the development of embedded computer systems. Characteristics of these systems are that they must operate in real-time and are resident in computer configurations where cost, size and weight constraints demand highly optimized programs.

3. The Compiler should provide comprehensive source language processing facilities. Although many of the functions listed below could be provided by separate tools which either process the original source or an abstract representation of the program, all of these functions may be incorporated as an integral part of the Compiler with little or no impact on compiler development cost, size, or performance. The functions that shall be incorporated are:

/ 5

a. Source reformatting, also known as "prettyprinting"

b. Static statistics collection and reporting

c. Program flow description

d. Run time support for dynamic statistics collection, performance tuning, path entrance verification, environment simulation and data recording/reduction tools

e. Recording of name usage from library units for subsequent production of a concordance listing by the Linker

f. Helpful aids listings including side-by-side assembly, cross-reference, statistics, diagnostics, and compilation summary.

4. The Compiler is to be easily retargeted and rehosted although these characteristics should not compromise target code efficiency

5. The Compiler should compile within 256K bytes of memory, but should make use of additional resources to increase its capacity and performance.

6. The Compiler must be very reliable and facilitate its own maintenance.

SECTION 2 - APPLICABLE DOCUMENTS

The following documents form a part of this specification to the extent specified herein.

2.1 PROGRAM DEFINITION DOCUMENTS

1. Reference Manual for the Ada Programming Language, July 1980.
2. Requirements for Ada Programming Support Environment, "STONEMAN", February, 1980.
3. Statement of Work, Contract No. F30602-80-C-0292, 80 Mar 26.

2.2 INTER-SUBSYSTEM SPECIFICATIONS

4. System Specification:.
5. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.
6. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.
7. Volume 3, Computer Program Development Specification for CPCI APSE Command Language Interpreter.
8. Volume 4, Computer Program Development Specification for CPCI MAPSE Configuration Management System.
9. Volume 6, Computer Program Development Specification for CPCI MAPSE Linker.
10. Volume 7, Computer Program Development Specification for CPCI MAPSE Editor.
11. Volume 8, Computer Program Development Specification for CPCI MAPSE Debugger.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

12. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.
13. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

14. Proceedings of the ACM-SIGPLAN Symposium of the Ada Programming Language, Dec. 9-11, 1980.

15. DIANA Reference Manual, 20 January 1981.

16. Rationale for the Design of the GREEN Programming Language, March 15, 1979.

17. Formal Definition of the Ada Programming Language, November 1980.

18. Ada Compiler Validation Implementers' Guide, October 1, 1980.

19. Sorting and Searching, Knuth, March 1975.

20. TCOL-Ada, Revised Report on an Intermediate Representation for the Preliminary Ada Language, Brosgol, et al, 15 Feb 80.

21. The Charrette Ada Compiler, Lamb, et al, Oct. 80.

22. An Informal Introduction to AIDA, Dousmann, et al, Nov. 25, 80.

23. AIDA Reference Manual, Persch, et al, Nov. 11, 80.

24. JOCIT/J3 Project Workbook.

25. J73 JOVIAL Project Workbook.

26. Depth-First Search and Linear Graph Algorithms, R. E. Tarjan, SIAM Journal of Computing 1:2, 1972.

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section describes the functional requirements and characteristics of the Configuration Item identified as the Ada Compiler.

3.1.1 General Definition

The Ada Compiler accepts as input a source object written in the Ada Language as defined in the Reference Manual for the Ada Programming Language, July 1980, and produces a relocatable object for one or more computers.

The Compiler is to operate initially on the IBM 370. The Compiler shall be written in Ada and a major design goal is that the Compiler shall be portable to other host computers and be easily retargetable, in particular, to embedded computers. To demonstrate the Compiler's adaptability, the Compiler shall be retargeted to the Interdata 8/32 and rehosted, by compiling itself, to the 8/32 under the OS/32 Operating System. The Compiler shall support, and permit selection from, multiple targets.

The Compiler design shall minimize the host computer system resources required for a compilation and permit utilization of available memory to increase compilation speed and input program limits.

The Compiler shall consist of multiple phases operating on various representations of the source program. The design shall include several global optimization passes. An innovative concept to be used to enhance the optimization effectiveness is to preprocess the IL produced by the front-end to incorporate machine dependencies into the program representation. Included in these dependencies will be such information as register vs. stack guidance, number of registers, calling convention expansions, addressing requirements, loop control orientation, etc. The intent is to supplement the information in the IL to orient the global optimization toward the selected target while retaining the machine independence of the major optimizing phases.

3.1.2 Peripheral Equipment Identification

The Compiler is to operate on the IBM 370 under the VM operating system and on the Interdata 8/32 under the OS/32 Operating System.

3.1.3 Interface Identification

The Compiler interfaces are identified as the ACLI, the KAPSE Data Base System (KDBS), the KAPSE Framework (KFW), the Editor, the Linker, and the Debugger. Figure 3-1 illustrates the relationship of the Compiler to the rest of the MAPSE.

3.1.4 Function Identification

The Compiler is functionally organized to permit the processing of an Ada program as a series of sequential phases that operate on various memory-resident tables or intermediate files representing the program at successive stages of the compilation. The functional phases are identified as follows:

EXEC	-- the resident compiler utilities
LEX	-- the lexical analyzer
LIBI	-- the Program Library processor
RESANL	-- the resolver and semantic analyzer
ALL	-- the data allocator
CHECKS	-- the constraint processor
TOP	-- the target computer IL transformer
FLOW	-- the flow analyzer
OPT	-- the main optimizer phase
LOOPER	-- the loop optimizer
CODE	-- the code generator
POST	-- the post code optimizer
ASM	-- the assembly lister and object formatter
XREF	-- the cross-reference processor
LIBO	-- the Program Librarian

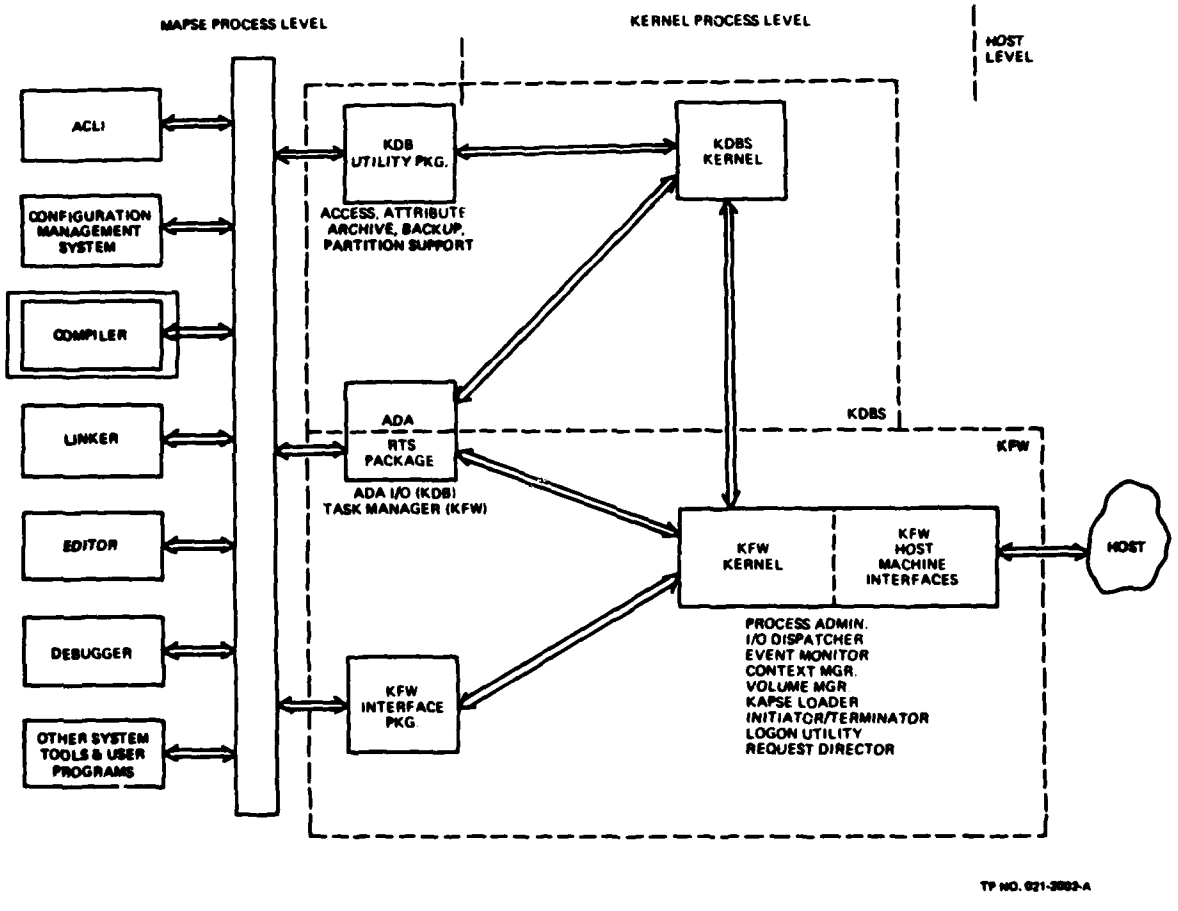


Figure 3-1. Interface Diagram

3.2 FUNCTIONAL DESCRIPTION

This section describes the functions of the Compiler, the program and interfaces identified above, and the utilization of input/output by the Compiler.

3.2.1 Equipment Description

The Compiler shall accept inputs and produce outputs in a device-independent fashion. The design shall permit compilation of a non-trivial Ada program with at least 50 executable Ada source statements in at most 256K bytes of memory.

3.2.2 Computer Input/Output Utilization

The Compiler utilizes the Ada Standard I/O Package to perform all of its file operations. The files utilized by the Compiler are described below:

Source	-- contains Ada program source or include files
Token	-- contains lexical tokens after LEX
Cref	-- contains cross reference information
ILs	-- several intermediate program representations
Code	-- contains code files from CODE and POST
Libs	-- contains the Program Libraries
Lrefs	-- contains label/proc/func references
Rel	-- contains the relocatable object
Listings	-- contains various Compiler listings
Error	-- contains diagnostic messages

3.2.3 Computer Interface

The Compiler has no direct hardware interfaces with the host computer but must understand inherently the target instruction repertoire and the addressing capability in at least the TOP, CODE, and POST phases. The remaining phases will draw any target dependent information required for their processing from tables which describe various target computer parameters. This information will be from two sources -- package SYSTEM and from Compiler tables whose target-dependent entry index is selected based upon a Compiler option used for selecting the target.

3.2.4 Program Interfaces

The Ada Compiler shall operate as an executable program under KAPSE. It may be invoked by the ACLI directly in response to a command or, may be invoked by any MAPSE tool or program, by a call with the INTERFACE pragma indicating a process. Figure 3-2 illustrates the data flow through these interfaces.

The method of invocation shall be transparent to the Compiler; its operation is identical in either case. The command parameters that include the identification of the input/output object names to be used for the compilation and the compilation, options shall be parsed by the Initiator (see Initiator in ref. 10) according to the Compiler main program unit-spec and initialized into heap space. A normal call is then made to the main program and compilation begins.

The Compiler shall utilize Ada Standard I/O for its file operations and shall operate in a device-independent manner. As with all tools, the mode of user connection (e.g., batch, interactive, etc.) shall be transparent to the Compiler.

Except for the KAPSE I/O interface, the Compiler interfaces are embodied in the objects read and written and in the code produced for interfacing with the Ada run-time support routines. The object interfaces are:

1. Source objects - Ada compilation unit source, INCLUDE source and the Compiler output listing objects shall all be represented in the standard text object format produced and accepted by the Editor.

   Integer and fractional line keys shall be accepted on source lines and shall be displayed on the source listings and used throughout the Compiler to cross-reference to the source statements. In absence of any keys on the input source object, ordinal line numbers shall be assigned by the Compiler and used for source referencing.

   Additionally, text objects may contain hidden attributes identifying changed or deleted lines since the previous version.

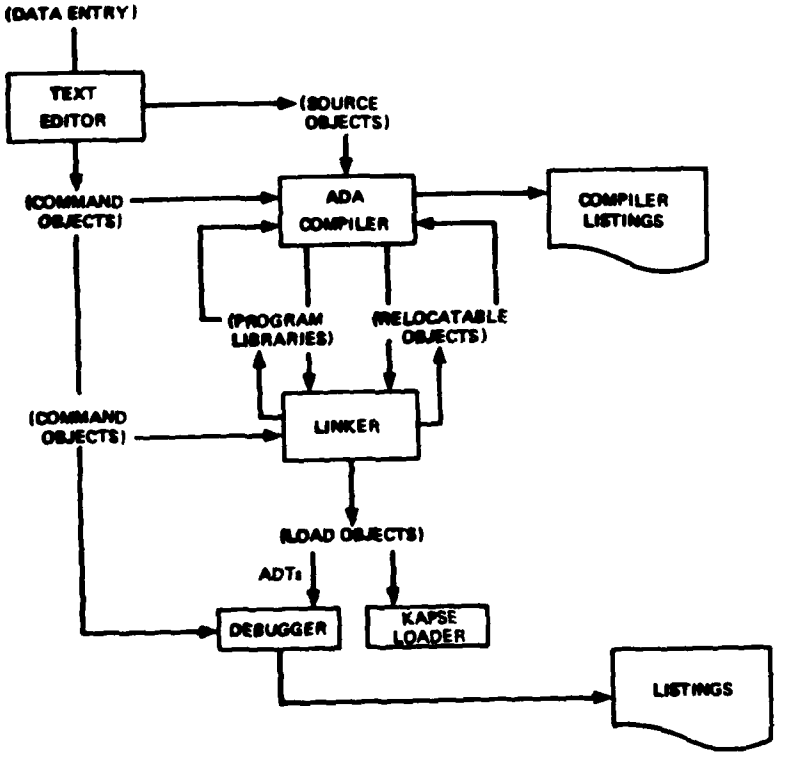


Figure 3-2. Data Flow

154

The Compiler shall display an indication of these changes in its source listings.

Each form of listing produced by the Compiler shall contain a unique character sequence in a fixed position on the listing line. This "listing code" may be used to identify and segregate listing types for quick examination of a compilation listing.

2. Relocatable Objects - The Compiler shall produce a relocatable object in MAPSE standard target-independent format. This object shall be read by the Linker. The format and contents of a relocatable object is described in Appendix D.

3. Program Libraries - During compilation of an Ada program, the Compiler shall access from, and create information for, a Program Library. This information, called "unit-specs", shall describe the visible compilation unit attributes. Library objects serve as the retention media for Compiler-to-Compiler, Compiler-to-Linker, and Linker-to-Linker communication. The format and contents of Program Libraries are defined in Appendix E.

Compiled programs shall contain calls generated by the Compiler to run-time Ada support routines to implement various Ada features such as tasking, exponentiation, string handling, space management, etc. The logical relationship of the compiler with the other MAPSE tools is shown below:

3.2.5 Function Description

As the one component of the MAPSE tool set that is required to parse and analyze Ada source programs, the Compiler is in a position to collect information about the program and to perform certain functions as a simple, inexpensive compilation by-product that would otherwise require substantial and largely redundant software. Furthermore, since the Compiler must be used during the course of program development, certain administrative functions may be performed by it unconditionally, possibly unbeknownst to users, that would otherwise require some additional level of control or conventions to achieve. Accordingly, the Compiler shall incorporate certain of these functions to reduce overall APSE development costs and to insure that the MAPSE system when delivered will incorporate those capabilities available in existing systems which, if lacking, might create a first impression of Ada and its support software that would be detrimental to its continued usage. These additional facilities are identified below.

3.2.5.1 Source Reformatting

The Compiler shall produce on option a reformatting representation of the input program. This representation shall be selectable in source object form as well as in listing from . The structure of the program as indicated by declarations, program flow statements, procedure bodies, and BEGIN-END blocks shall be made clearly visible by source indentation. Nesting levels shall be indicated for each nest/unnest association. Source included via the INCLUDE pragma shall be displayed as selected. Source positional numbering used by the Compiler to relate various stages of the intermediate and final program representations to the source shall be noted on the listing. This same numbering system shall allow the user to express program position to the debugger.

3.2.5.2 Statistics Collection

Although the SOW explicitly requests collection and reporting of program and compilation statistics, the Compiler shall support a facility to implant runtime code or hooks to permit collection of dynamic statistics and timing information. Included in the facility is path entrance verification. The

intent is to assist users in both program checkout and performance tuning. Additionally, the Compiler shall permit the collection of compilation and error frequency statistics for administratively monitoring the error proneness of the Ada language features.

3.2.5.3 Program Flow Description

As a fallout of the flow analysis required for comprehensive optimization, a program flow graph description shall be produced to accomodate path entrance statistics as well as automatic flow documenting tools.

3.2.5.4 Language Modularity

In recognition of the fact that the Ada language is a very new development and is pioneering new facilities in a higher level language such as generic procedures, comprehensive exception handling, built-in tasking and non-trivial operator and procedure overloading; a concerted effort will be made to provide flexible language analysis algorithms and hence accomodate changes to the language in the advent that ambiguities or omissions are discovered.

3.2.5.5 Retargetability

To promote retargetability, a standard relocatable object format shall be developed. This format shall be independent of the target computer word size. In addition, to minimize cost, a standard relocatable object formatter shall be designed for the Compiler and linker and delivered as an additional Library utility for use by later APSE programs that need to construct relocatable objects. Similarly, a standard assembly lister capability shall further reduce the cost of future retargetings.

3.2.5.6 Listing Aids

As interactive program development increases, the need exists for efficient mechanisms for the perusal of listings at conversational terminals. Toward this end, the various Compiler listings shall be distinguished from each other by a listing code convention that will allow the user to easily locate and list particular kinds of listing output. This facility shall support the scanning of a compilation listing to determine the existence and cause of any diagnostic messages.

3.2.5.7 Environment Simulation Support

A consideration in the design of program and package interface files and the debug tables produced by the Compiler shall be to satisfy the anticipated requirement for sufficient description of a program's data base to permit the development of environment simulation programs and data recording/reduction tools.

3.2.5.8 Object Version Genealogy

One of the important functions required during the development of large systems is one of configuration management. The SOW indicates that the MAPSE will assist managers in controlling and identifying configurations. In support of this function, the Compiler shall include in the generated relocatable object tracing information that shall describe the derivation of the object program. Included in this information shall be the version of all input files (original source, INCLUDEd source, Library units) as well as that for the Compiler itself.

3.2.5.9 Compilation Order Validation Information

In conjunction with the above described information, the Compiler shall support the Linker as necessary to permit detection of attempts to combine object programs that possess incompatible interface specifications. This can arise because different descriptions were used or, for example, if a called procedure's specification was changed since some caller was last compiled.

3.2.5.10 System Usage List

When a program is compiled, each name referenced from a Library unit shall be recorded in the relocatable object. An option of the Linker shall be to process this file containing the output from each compilation in a system, purging obsolete compilation data, and sort it in a user specified manner. Sorted and printed, this data will provide a handy concordance of the usage of Library unit names by compilation.

3.2.5.11 Symbolic Maintenance Facilities

The Compiler shall have an extensive capability to monitor symbolically its own operation. Symbolic dumps of the Compiler's data base shall be a built-in function of the Compiler. The Compiler shall provide the ability to trace its operation. These maintenance functions shall be controlled dynamically to permit dumps and/or traces within selected phases and during the processing of selected program statements.

3.2.5.12 Reliability

The Compiler must be reliable. Whenever there is a switch from an existing language or manner of operating to new techniques, the effectiveness of the new system is of utmost importance.

3.2.5.13 Maintainability

Several factors affect the maintainability of a large system such as an Ada Compiler. Perhaps most importance of these is that all components of the system--or compiler--are designed with a consistent philosophy. This will be done. The Ada language shall be exploited to "package" the external data base of the Compiler, to encapsulate symbol table entry addition and deletion, and to modularize the Compiler components. As the Compiler will be implemented in Ada, the Compiler shall be used as a substantial regression test of its viability.

The Compiler shall contain an integrated symbolic debugging facility to assist maintenance personnel in the location of Compiler malfunctions, to permit temporary corrections and to validate changes. This facility will be controllable at several levels of granularity. Traces and data dumps may be requested at the highest level by command option. At the finest level, traces and dumps may be selected by individual phase and statement using maintenance pragmas. Additionally, the Compiler shall interface with the symbolic debugger by the generation of special debug tables to further improve the maintainability of the Compiler.

16-V

3.2.5.14 Performance Characteristics

The Compiler shall be designed to compile a nontrivial Ada program in 256K bytes of memory at a compile speed in excess of 1000 statements per minute.

3.2.5.15 Human Performance/Engineering

The Compiler shall be user-oriented. The diagnostics message facility will be selected to accurately pinpoint the location of erroneous source and to identify the offending constructs. Listings shall be clearly cross-reference to original source position. Optimizations, which are often confusing to novice and sophisticated users alike, shall be described with helpful messages in the assembly listings.

Recognizing the potential impact of the Ada compilation order requirements, the Compiler shall inform the user of potential and real order violations.

3.3 DETAILED FUNCTIONAL REQUIREMENTS

This section describes in detail the functional organization of the Ada Compiler. The Compiler comprises a set of memory-resident utilities referred to collectively as the EXEC and fourteen functional phases. The phases, in general, gradually phase by phase transform an Ada program from source object form into a relocatable object. Some of this transformation is performed by processing data retained in memory between phases; the major portion of the transformation is performed by reading a file representation of the program, traditionally called IL for intermediate language, massaging the program representation to parse the various language constructs, organize the program description first for optimization and then for code generation, mapping the program from a tree form onto the target computers instruction set, and, producing the relocatable object suitable for linking by the MAPSE Linker and eventually executing.

The compiler is very modular in structure and the arrangement of the various phases into overlay segments is quite flexible to accommodate a wide spectrum of host memory resources. The organization pictorially is presented in Figure 3-3.

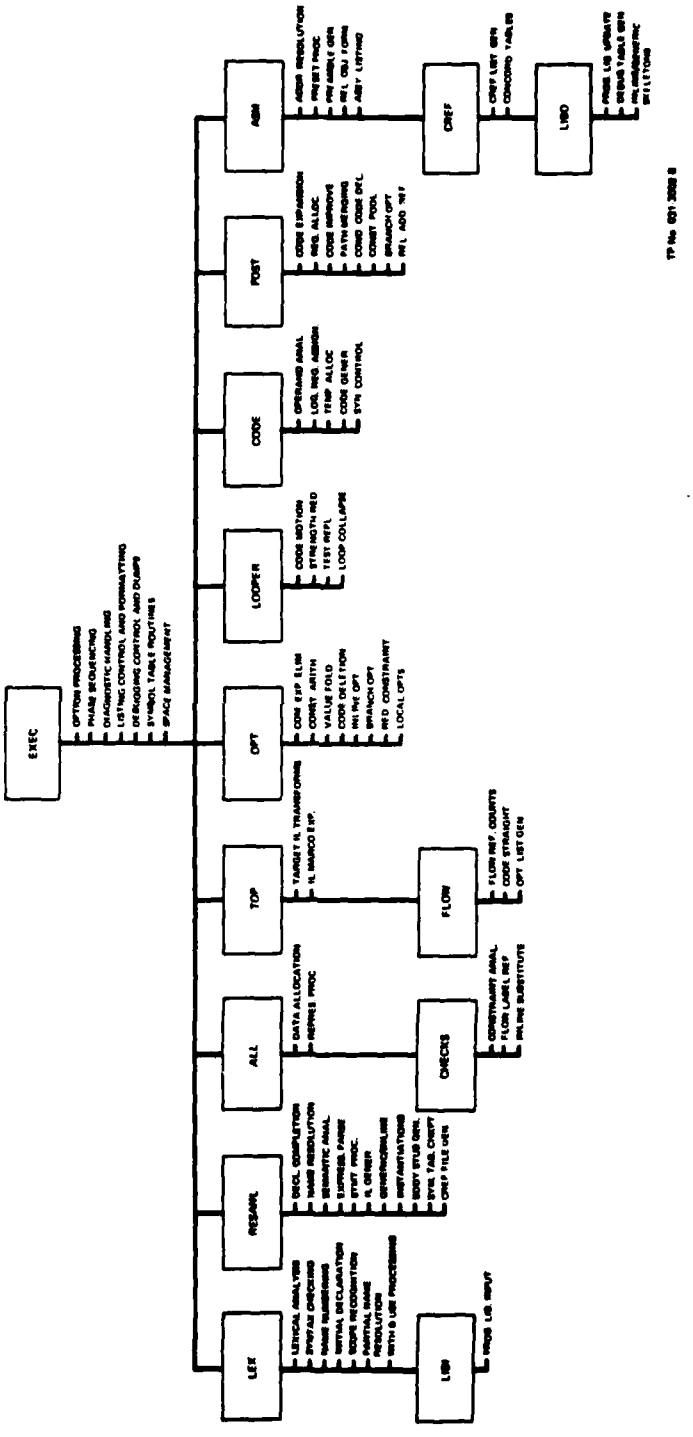


Figure 3-3. Compiler Structure

The Compiler shall be written entirely in the Ada Language and will be host independent except for certain table representations which will be hand allocated to reduce the memory requirements of the Compiler. All interface data declarations will be in packages. To the extent possible the declarations will be described using compile formulas based upon the target attributes.

3.3.1 Compiler Executive - EXEC

The Compiler is structured as a tree with a resident root section and several levels of branches or phases. The root section is called the EXEC and it contains the routines and data that are required throughout the compilation process. Although a small section of the EXEC may be thought of as the Compiler main program, the EXEC is composed largely of sets of completely independent procedures.

3.3.1.1 Inputs

The Compiler is invoked with a set of options indicating the processing to be performed and a list of object names identifying the source, libraries, listing, and relocatable objects to be used for this compilation.

3.3.1.2 Processing

Functions that are supported by procedures in the EXEC are:

1. Sequencer - Once the compile options have been determined, the various phases of the Compiler required for this compilation must be loaded in the proper order and executed.

2. Space Management - The Compiler has special requirements for maintaining the Compiler resident data base; these functions shall be concentrated in the root. Any requirement for cross-phase heap space shall be handled through these routines.

3. Listing Routines - A standard set of listing generation routines shall be included as part of the EXEC. Included among the functions supported shall be title and subtitle settings, line printing (including pagination control), listing coding, page ejecting and line spacing, and miscellaneous formatting routines.

4. Diagnostic Routines - A common set of routines shall be used by all Compiler phases for the reporting of Compiler diagnostics. This routine shall utilize a phase dependent message description array, shall permit insertions into the message of source names or other character strings, shall interface with the command level message brevity control, shall save diagnostics for later sorting by source line key for interspersing or separate printing as selected, and shall maintain counts for collection of error frequency statistics and reporting in the compilation summary listing.

5. Symbol Array Package - All posting of entries into the symbol array and searching for prior entries are accomplished through these common package routines.

6. Debugging - The Compiler shall contain extensive symbolic debugging facilities for assisting in its own development and maintenance. These routines shall be included as an integral part of all production compilers but shall be arranged in a fashion so as to not increase compile space requirements except when enabled.

3.3.1.3 Outputs

A compilation control record shall be created by the EXEC which will contain the list of objects to be processed and/or created and the options used to control the various phases' processing.

3.3.2 Lexical Analysis - LEX

LEX is the lexical analysis phase of the Compiler, its major function is to read the Ada source program and encode the entire program as a sequential set of tokens and write a token file. In addition, LEX performs scope recognition and begins building the symbol table structure, builds the Name table, and builds the Context Specification table and the Unresolved-Name table.

3.3.2.1 Inputs

The input to LEX is the Ada source program being compiled. The elements of the predefined language environment (e.g., package STANDARD) are an implicit input to LEX but they are not processed as normal Ada program source input.

3.3.2.2 Processing

The major functions of LEX are outlined below.

3.3.2.2.1 Source to Token Transformation

The entire source program is read, all lexical units are recognized, converted to tokens and written to the Token file. The following Token types are defined:

1. Reserved-Word Token - Every Ada reserved word shall be converted to a reserved word token; the reserved word shall be specified with a reserved word number. For purposes of the token file, special character delimiters and compound symbols (e.g., ';' '+' '-' '/' '**' '<<' '<=') shall be treated as reserved words.

2. Attribute Token - The predefined language attributes ('BASE, 'SIZE, 'FIRST, 'LAST, etc.) shall be converted to attribute tokens, the particular attribute being specified with an attribute number.

3. Line Number Token - A line number token shall be output preceeding the first token of each line. The record key of the associated line shall be output as the line number, if the source program has no line numbers, they shall be created starting at 1 in increments of 1.

4. Name Token- All user names (or identifiers) shall be converted to Name tokens, which will contain a name-number to uniquely identify every name in the compilation unit. With the name-number and the Name table the actual character string of every name can be retrieved.

5. Literal Tokens - Literals shall be converted to a canonical form and passed in variable length Literal tokens. The following types of literal tokens shall be used:

   a. Numeric integer

   b. Numeric real

   c. Character literal

   d. Character string literal.

6. Resolved-Name Tokens - For all references that are fully resolved in LEX, the name is replaced with a resolved-name token which contains a symbol table pointer to the resolved symbol table attribute entry. The actual name of the item is still available through the name-number and the Name table.

3.3.2.2.2 Name Table

A Name Table shall be built that will contain every unique name (or identifier) used in the compilation unit as an actual character string. Each unique name shall be given an integer name-number assigned from some initial value in increments of 1. Associated with the Name Table shall be a hash table and a Name Pointer Table. Given a name-number, the actual name and every symbol table attribute entry associated with that name can be accessed.

3.3.2.2.3 Skeleton Local Symbol Table

LEX shall perform partial syntactical analysis - sufficient to recognize the declarations of objects, types, subtypes, subprograms, packages, tasks, blocks, loops, exceptions, and renaming declarations. Symbol table entries shall be posted for each declaration - since the attributes of these entries are, in general, unknown in LEX, the attributes shall not be filled in and only skeleton symbol table entries shall be built. The main purpose of the skeleton symbol table in LEX is so that resolution may be performed on all simple local name references.

3.3.2.2.4 Unresolved Name Table

During the lexical analysis process, name resolution is attempted on all user identifiers using the current local symbol table. Where an unambiguous resolution can be made, a resolved-name token is output for the reference (no semantic check on the correctness of the reference is possible in LEX). Where an unambiguous reference cannot be made, the name is output as an unresolved name token and the user name including all qualifiers, is entered in the Unresolved Name Table. At the end of LEX, the Unresolved Name Table shall contain a complete list of all user names that are potentially defined in the compilation units specified in the context-specification (the "WITH unit-names" and also the "NEW generic-names") of the program being compiled.

3.3.2.2.5 Context Specification Table

The context specification table, built by LEX, shall contain the name of every Library unit mentioned in a "WITH clause" and the name of every generic unit being instantiated ("NEW generic-name"). This table shall be used by LIBI for extracting from those Library units all possible declarations and specifications need by this compilation.

3.3.2.3 Outputs

The outputs of LEX are:

1. Token File
2. Skeleton Symbol Table -- local declarations
3. Name Table
4. Unresolved Name Table
5. Context Specification Table.

3.3.3 Library Specification Input - LIBI

LIBI is a small phase and runs between LEX and RESOLV - lexical analysis and semantic analysis. The function of this phase is to extract from the Library units specified in the context-specification (the "WITH unit-names") all those declarations and specifications that are potentially required by the program being compiled and to post these Library unit declarations and specifications to the symbol table.

3.3.3.1 Inputs

The inputs to LIBI are the Program Library, the Context-Specification Table and the Unresolved Name Table - these latter two tables are produced by LEX.

3.3.3.2 Processing

The Context-Specification Table contains the name of each Library unit ("WITH unit-name") that must be accessed. Each Library unit is processed against a sorted Unresolved Name Table and all those declarations and specifications that are potentially needed to complete the resolution process are extracted from the Library unit and posted to the symbol table.

171

Note that this process may bring in unrequired Library unit specifications but among those extracted shall be all those needed to complete the resolution process.

3.3.3.3 Outputs

The output of LIBI is an updated symbol table. The skeleton symbol table built by LEX shall contain attribute entries for all those entities defined by the program being compiled (the contents of these skeleton entries will be incomplete).

LIBI shall extend the symbol table with every potentially referenced unresolved entity defined in one of the Library units specified in the context-specification ("WITH unit-names").

3.3.4 Resolution and Semantic Analysis - RESANL

RESANL performs the syntactical and semantical analysis of the program, completes the building of the symbol table, completes the name and operator resolution process, performs overloaded resolution and generic instantiation and optionally produces the cross reference file.

3.3.4.1 Inputs

The inputs to RESANL are the following:

Token File - The token file is produced by LEX and is an encoded representation of the full source program in token form.

Symbol Table - The symbol table contains the attribute entries of all the ADA entities defined or required by the program being compiled. The symbol table entries of all those entities defined by the program being compiled are as yet incomplete - their structure is present only.

3.3.4.2 Processing

RESANL rereads the entire source program, which is now in Token form, completes the symbol table entries for all local declarations, performs resolution and semantic analysis on each statement and produces the intermediate representation of the source program (IL). The major functions of RESANL are outlined below.

Declaration Parsing

All declarations are syntax checked and fully processed. Final name resolution is performed and the attributes of the skeleton symbol table entries posted by LEX are completed. Errors are diagnosed and suitable defaults chosen where possible.

Declarations within generic units may not be fully resolved at this time since they may depend on generic parameters. However, the symbol table entries for such declarations will be structurally complete since they will reference the generic formal parameters.

All declarations are optionally written to the cross reference file.

Note that since certain declarative attributes (e.g. bounds or ranges) may include expressions and may not be compile-time constant, general expression analysis is required during declaration processing.

The constant values of constant declarations will be posted to the symbol table.

Initial value expressions for local data will be converted to IL as assignment statements with the 'REPL' operator.

Variable expressions specifying array bounds will be converted to IL as assignment statements to the array's dope vector.

Constant initial values of non-generic package data will be written to the preset file - this data does not require dynamic initialization at elaboration time but is elaborated at load time.

Constant initial values of generic package data will be converted to IL as special preset assignment statements with the 'PRESET_REPL' operator. At instantiation of generic packages the special preset assignments will be converted to normal preset file entries.

Statement processing

All statements are syntactically checked and name resolution is performed. Expressions are processed by the expression analyzer (which performs name and operator resolution on the expression) semantic checking is performed and the statement is converted to IL for output.

1/5

Expression processing

Expression analysis is the most significant processing performed in RESANL. Expressions are syntactically checked and an expression tree is built. The expression tree is scanned and name resolution is attempted ignoring context. User identifiers are either unambiguously resolved (only one possible visible definition - it may or may not be semantically valid), ambiguously resolved (more than one possible definition requiring further analysis in context) or unresolved (undefined identifier - immediate error).

Operator resolution is not performed on this first scan of the expression tree. If any of the user identifiers are undefined, diagnostics will be issued and the statement will be deleted; otherwise, further scans are made up and down the expression tree attempting resolution on all operators and overloaded functions and literals - here use is made of the context of the expression along with the possible multiple resolutions found on the first scan.

If unambiguous resolution cannot be accomplished (either the ambiguity of some references could not be resolved or some of the operators were undefined), the errors will be diagnosed and the statement will be deleted.

The resolution of procedure and entry calls will be handled very much like function resolution except that the call has no context to resolve an ambiguity.

After each statement is successfully resolved and checked semantically, it is reformatted as IL and output to the IL file. The cross reference file is optionally written for each reference.

Generic Instatiation

Generic instantiation is performed in RESANL. For each generic instantiation the generic parameters are resolved and identified. The generic specification of the generic declaration is accessed from the generic declaration file or from the Program Library - generic specifications may have been extracted from the Library unit specifications by LIBI or may have been created by generic declarations processed

174

previously in the program being compiled. The generic specification contains symbol table entries for defined and referenced items and contains structurally complete IL for all statements within the generic declaration. The generic specification is read in, generic actual parameters are substituted for generic formal parameters, any new symbol table entries required by the generic are posted and the statements of the generic declaration are processed through the normal analysis and resolution procedures in RESANL.

Generic Declaration

Special processing is required for generic declarations. The result of the generic unit may have to be eventually written to the Program Library file as part of the program's Library unit specification or the generic unit may be instantiated later in the program being compiled. While processing a generic declaration a special mode will be in effect - the main difference between "generic-declaration" mode and normal mode is in the resolution process where certain references may remain ambiguous until generic instantiation.

When the processing of a generic declaration unit is complete, the IL and symbol table needed by the generic will be optionally written to the generic declaration file for later instantiation in this program or for use in writing the Library unit specifications for the program.

3.3.4.3 Outputs

The outputs of RESANL are the IL file, a completed symbol table and an optional generic declaration file. In addition, RESANL also optionally outputs a cross reference file (as a function of the cross-reference compiler option).

The cross refrence file is written by RESANL on a compiler option and contains an entry for every defined and referenced user identifier in the program, for all object references the entry indicates whether the reference was an assignment to the object or a use of the object. The cross reference file is used to produce the cross reference Set/Use listing.

The cross reference record is defined below.

```
type CR_REF_TYPE is (USE, SET, DECLARE); type CROSS REF is
 record
  CR_ITEM:  SYMPTR: -- symbol table pointer
  CR_TYPE:  CR_REF_TYPE; -- used, set or defined
  CR_LINE:  SRKEY; -- line number
  end record
```

3.3.5 Allocator - ALL

3.3.5.1 Inputs

Symbol table - resident - (See Appendix A)

3.3.5.2 Processing

The allocator is responsible for mapping the various data items which have been declared in the source program onto target machine storage locations. The allocator must honor address specifications and record type representations as well as the pack pragma. (The means by which the allocator is informed of the existence of a pack pragma is through a flag in the symbol table).

The allocator must determine the representation to be used for the various types which appear in the symbol table. This includes determining the size of each item which has no explicit range given in a record type representation and no explicit length specification.

Similarly, the allocator must assign relative storage locations to the various objects in the symbol table, honoring any alignment clauses which may be present. The addresses assigned by the allocator are relative to the start of some block of data, (for example, the local stack frame).

The allocator must take into account the effect of variants on both the packing of items into storage units and the computation of addresses. Alignment requirements imposed by the target machine should be taken into account, also.

The allocator is target dependent to a certain extent, but will be parameterized according to the storage unit size, address granularity, alignment requirements and data types available, to maximize retargetability.

176

3.3.5.3 Output

Symbol table - (See Appendix A) - Location specifiers and position fields filled in.

3.3.6 Constraint Analysis Processing - CHECKS

CHECKS performs constraint checking analysis - creating constraint IL for all the required CONSTRAINT_ERROR and NUMERIC_ERROR checks. CHECKS is not concerned with the other Ada exceptions since they are not handled directly in the generated code. CHECKS reads the IL file produced by RESOLV and writes a new IL file. In addition, CHECKS produces the Flow Reference List file (FRL).

3.3.6.1 Inputs

The inputs to CHECKS are the IL file produced by RESOLV and the resident symbol table.

3.3.6.2 Processing

CHECKS reads the entire IL file and processes each statement in turn. Constraint IL shall be inserted where appropriate and the IL file is rewritten.

The SUPPRESS pragma's and Compiler options shall be honored to suppress constraint checking.

Constraint checks shall not be inserted where the error cannot occur; e.g., I := J; No range check shall be inserted if I and J are of the same type or if J's range is a subset of I's range. However, duplicate constraint checks shall in general be inserted by CHECKS - these shall be removed by the optimizer in its normal common sub-expression analysis, folding could eliminate further unnecessary constraint checks.

The required constraint checking analysis is outlined below.

3.3.6.2.1 CONSTRAINT_ERROR Checking

Assignment Statements - Constrained Value Computation - A range check is conditionally inserted at every assignement statement or evaluation of an explicitly constrained value; this includes initailizations, return

177

statements, qualified expressions, type conversion, and parameter associations.

Array Referencing - Bounds checking shall be conditionally inserted on every array reference.

Array Assignments/Expressions - Array size checks (number of elements) shall be conditionally inserted at every array or array slice assignment or expression.

Access Variable Checks - Access value checks against the null value shall be conditionally inserted at every use of an access variable for object referencing.

Discriminated Component Referencing - Discriminant value checks shall be conditionally inserted at every reference to a component of a discriminated record.

3.3.6.2.2 NUMERIC_ERROR Checking

For the numeric error exceptions (divide by zero or hardware overflow) CHECKS does not insert explicit constraint IL but rather marks each IL operator which could potentially cause such an error.

On a target machine where the hardware does handle divide by zero or computation overflow with a trap (i.e., where explicit instructions were required to test for these conditions) and where such checking was required, the code generator shall use the numeric error flag in the IL operator to control the generation of code to perform the overflow checking.

Numeric error checking is being handled in this way for two reasons - so as not to unduly increase the size of the IL and so as not to burden the code generator with the analysis required to determine where hardware overflow needs to be tested.

3.3.6.2.3 Flow Reference List File

The flow reference list file is written by CHECKS as it reads the IL file; the file contains an entry for every location in the program to which control might be transferred (definition points) and an entry for each

possible transfer of control to such locations (reference points). The record contains both the scope of the entity and the entity being referenced or defined. Definition points result from labels, procedures, functions, and task entries. Reference points result from 'GOTO's (explicit and implicit), procedure calls, function calls, and entry calls.

3.3.6.3 Outputs

The output of CHECKS is a updated and rewritten IL file and the Flow Reference List file. The structure of the Flow Reference List file record is shown below:

```
type FLOW_REF_TYPE is (REF, DEF);  -- reference or definition
type FLOW_REF_REC is
  record
    FRL_SCOPE:  SYM_PTR;  -- symbol table pointer of most
                          -- inclusive scope
    FRL_SYM:  SYM_PTR;  -- entity being reference or defined
    FRL_TYPE:  FLOW_REF_TYPE;
  end record;
```

3.3.6.4 Special Requirements

The constraint checking analysis could logically be done in RESANL as the IL statements are output. However, while much of the constraint analysis is straightforward some of the analysis - especially involving array and array slice expressions and assignments - is quite complex and the space requirements of a combined RESANL and CHECKS could be excessive.

3.3.7 Target Optimizer - TOP

This phase of the Compiler is used to tailor the IL to the target computer and is expected to greatly enhance the benefits derived from the target-independent global optimizer.

3.3.7.1 Inputs

TOP shall read the IL file produced by RESANL and make extensive use of the resident symbol array data. The format of this data is described in Appendices A and B.

3.3.7.2 Processing

The essence of the TOP functions is to read the IL produced by the semantic analysis phases and perform transformations on the IL to orient the IL to the target computer and hence the optimizations performed by the remaining optimizer phases. The form of the IL input to TOP is the saem as that produced by TOP. However, for any particular target computer, certain operators may never appear in the IL after the TOP phase. Conversely, some IL operators may only be produced by TOP.

Transformations that may be made by TOP depending on the target selected are described below:

1. Linearize subscripts to best utilize the target's indexing capability.
2. Combine shifts, scaling, and powers of two multiplication and division as appropriate to the number representation of the target machine.
3. Expand object references according to the target characteristics, including inserting explicit address constant references, address unit multiplications, extraction/deposit IL operators, observing displacement size constraints, etc.
4. Orient local data references depending upon the target machine's stacking mechanisms.
5. Tailor procedure calls and parameter passing to the conventions of the target.
6. Expand IL macros in a target-dependent manner.
7. Parameterize the IL and IL operator descriptor tables for the target machine. This parameterization will include defining the number and characteristics of the hardware registers.
8. Convert constraint checks and loop control into the target-specific operators for simple ifs, gotos, and assignments.

180

3.3.7.3 Outputs

The IL is written out in the same form as read and as described in Appendix B.

3.3.8 Flow Analyzer - FLOW

The primary difference between a cross-statement (basic block optimizer) and a full global optimizer, as described in the following sections, is that statements thath result in program flow or subprogram calls will restart the optimizer's data base. The processing that makes global optimization possible is the analysis of program flow and the collection of information regarding the existence of variable object references or assignments within the basic blocks connected by the flow. This function is called "flow analysis" and is performed by FLOW, the initial target independent global optimizer phase.

3.3.8.1 Inputs

The input data base used by FLOW are the FRL described in section 3.3.*.3 and the symbol array and IL described in the Appendices. The data created by FLOW are described below:

The out of scope label reference list (occuring in Ada only as a result of optimized case and raise statements), the variable set and used list for procedures and loops are simply an array of indices into the symbol array. A slice of the array corresponds to the range of a loop or procedure. The beginning and length of the slice is retained in procedure attribute entries or in the loop descriptor list (described below) as appropriate.

```
plrl:array(1..max_plrl) of sym_ref;  -- proc label ref list
prvl:array(1..max_prvl) of sym_ref;  -- proc ref variable list
psvl:array(1..max_psvl) of sym_ref;  -- proc set variable list
lsvl:array(1..max_lsvl) of sym_ref;  -- loop set variable list
type loop_list is
     record
     loop_start:il_ref;  -- IL index for beginning of loop
     loop_end:il_ref     -- IL index for end of loop
```


181

```
        lsvl_ix:integer;     -- lsvl index for initial set variable
        lsvl_no:integer;     -- count of variables set in loop
        loop_lab:sym_ref;    -- loop top label
        end record;
    lopl:array(1..max_lopl) of loop_list;  -- loop descriptor list
```

3.3.8.2 Processing

FLOW is separated into three separate functions; these functions are the computation of reference counts, code straightening and list generation.

3.3.8.2.1 Reference Count Computation

The initial function is to read the Flow Reference List (FRL) produced by CHECKS into memory and link all references to procedures and/or labels by containing scope. Once this linking is complete, each chain is followed to its end, nesting as necessary at a procedure call, and incrementing the reference count for any target declared in the scope of the chain being chased.

3.3.8.2.2 Code Straightening

Upon completion of the reference counting, FLOW shall read the IL, a scope at a time, and produce a straightened graph of the scope. This straightening of the program flow insures that the assumptions made by the main processing of FLOW are indeed valid. These assumptions are:

1. A forward branch encloses conditionally executed code
2. A backward branch encloses a loop.

The straightening algorithm employed is that described in "Analysis of Graphs by Ordering of Nodes", Earnest et al, Journal ACM, Jan. 1972. The algorithm performs the straightening by first building the flow graph description as described in Appendix E (see types block and path). The preferred ordering is then computed and then the IL is relinked in the straightened order. The relinking process does not recopy the IL but simply inserts branch, label and link operators as necessary to reflect the new order.

157

A short example will serve to illustrate the effect of this straightening. In the diagram below, the flow of a trivial program is presented on the left before straightening, on the right after straightening

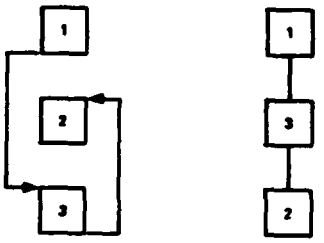


Figure 3-4. Code Straightening

In the graph on the left, block 2 appears to be conditionally executed whereas blocks 2 and 3 appear to be within a loop. Assignments in block 2 would prevent subexpressions using the changed operands from being found common from block 1 to 3 when in fact no interference could result. The straightened graph would of course allow the optimizer to correctly recognize that block 2 did not exist on any path from 1 to 3. Although some direct benefit results from this reordering (the deletion of branches) the primary benefit accrues from improved optimization by the remaining phases.

3.3.8.2.3 List Generation

The primary purpose of this section of FLOW is the production of the lists utilized by the remaining phases of the optimizer. these lists are the Label Reference List (containing each program branch or call), the procedure

lists (containing the variables referenced or set by the procedure) and the loop description lists including the variables set within the loop. The bulk of the processing however is the actual flow analysis which drive the building of these lists. The flow analysis algorithm used is the evolution of the Linear Nested Region Analyzer (LNRA) scheme developed by CSC and first used in a production compiler for JOVIAL/J3; the most recent of which was the JOCIT/J3 System Compiler. This algorithm is quite fast and requires less data than most graph analysis algorithms, such as those of Tarjan [26]

The lists are produced by processing the IL, scope by scope, adding variables to the appropriate lists depending upon usage context. As a procedure entry is encountered, the present list indices are recorded in the procedures attribute entry. At exit, the count of the entries added to the various lists are recorded. At completion of the list, a procedure symbol table entry will appear as follows:

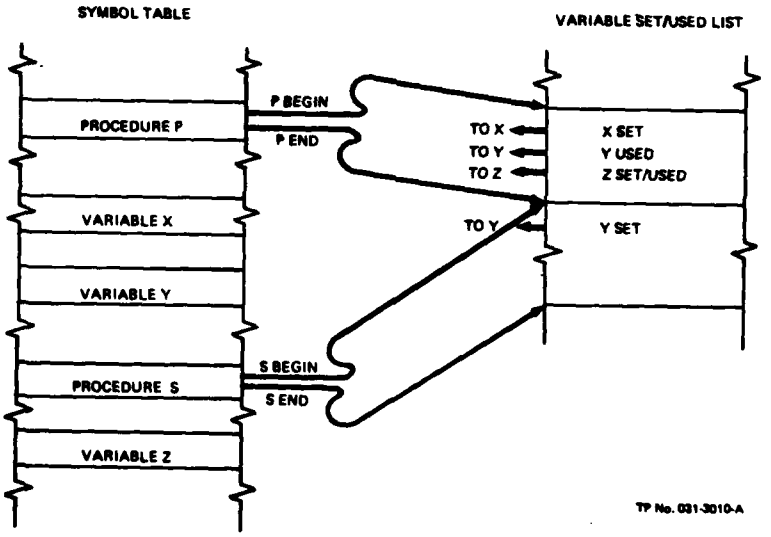


Figure 3-5. Procedure List Pointers

Similar processing occurs while processing labels. As a label reference is encountered, the reference count is decremented. When a label is encountered, the fact is noted in its attribute entry. If the reference count is not zero, backward branches exist to this label for all outstanding references. The current lsvl index is recorded. A reference to a label already encountered closes a loop. The loop descriptor is modified to reflect the loop closure and lsvl count is recorded.

3.3.8.3 Outputs

The output of FLOW is the IL as described in Appendix B, the symbol array described in Appendix A, the flow graphs described in Appendix E and the memory-resident lists described in 3.3.8.1.

3.3.9 Optimizer - OPT

OPT performs the major program transformations which result both in reduced program code space and execution time savings.

3.3.9.1 Inputs

The input to OPT is the symbol array and IL described in the Appendices and the lists created by FLOW and described in 3.3.8.1.

3.3.9.2 Processing

Operands are viewed as having a identifiable value throughout the OPT phase. This value may be known at compile-time to be a constant or to be undeterminable. When values are assigned, the variable assigned becomes a synonym for the value and hence improves the possibilities for common subexpression elimination. For example,

```
.... x+y  --reference to an expression
z:=x:          --z becomes a synonym for x
.... z+y; --expression need not be recomputed
```

Note that if in the above sequence x had been assigned a new value immediately after the assignment to z, the common expression recognition has not been affected. This "value folding" is the basis for these optimizations.

As each scope is entered, a unique value number is assigned to each variable. If the variable is constant or is initialized to a constant expression during the elaboration, the constant value is recorded as its synonym. When an operand is referenced, its present value number is posted to a search list. Each expression is also posted to the search list by creating a value number from its operator and present operand values.

As loop boundaries and forward branch targets are encountered, the search set must be pruned for the range of the enclosed region or intersected with joining paths.

The optimizations performed are:

Value Folding - Recognizing that an assigned variable is a synonym for the expression assigned.

Common Subexpresion Elimination - Recognizing that an expression has been computed previously and may be used without recomputing.

Constant Arithmetic - Performing at compile-time all expressions whose result values are computable because of constant operands. Constant conversions and relationals are included in this function.

Code Deletion - Code which cannot be reached (often caused by the recognition of relational tautologies or contradictions) shall be deleted. Unreferenced procedure code shall also be deleted.

Inline Procedures - Procedures referenced by a single call or where the inline expansion is smaller than the call code shall be expanded in line.

Branch Optimization - IF THEN GOTO shall be transformed in a manner to eliminate a conditional branch around a branch. Branches to a label occurring immediately in front of the label shall be deleted. Branches to branches shall be optimized to transfer directly to the final branch point; the intermediate branch shall then be deleted if it is now unreachable.

Miscellaneous - The local optimizations listed below shall be performed.

x*1 == x

x/1 == x

x*0 == 0

x/0 == diagnosed

x/x == 1

x+0 == x

x-0 == x

x**0 == 1

x**1 == x

x*2**y == x scaled y

x=x == true

x x == false

Redundant Constraint Checks - A constraint check shall be deleted when the optimizer can determine that the check is unnecessary because of previous checks or compile time determinable values and relations. This optimization shall be performed by extending the value folding concept and retaining value-range realtionships essentially as optimizer generated assertsions. These assertions shall be associated with the value numbers at assignements, as constraints checks are passed and by the derivation of values from the paths taken at if and case statements.

3.3.9.3 Outputs

The output of OPT is the IL files.

3.3.10 Loop Processor - LOOPER

The final global optimizer phase performs the analysis of loop structures, code movement from loops and dead variable analysis. Many of the LOOPER optimizations are oriented more towards execution tim savings rather than program space reduction.

3.3.10.1 Inputs

The LOOPER uses the IL written by OPT, the lists created by FLOW and the symbol array.

3.3.10.2 Processing

The functions of this phase of the Compiler are to analyze the loop structures of a program as described by the LOPL list prepared by FLOW and the context of blocks enclosed by the loop paths. The optimizations performed by this phase are code movement, strength reduction, test replacement, loop collapse and dead variable analysis.

3.3.10.2.1 Invariant Code Movement

Expressions computations will be moved from within loops when the operands are not modified within the loop.

3.3.10.2.2 Strength Reduction

Operators such as multiply and exponentation will be converted to addition and multiply, respectively, when the operands are iteratively incremented in a loop.

3.3.10.2.3 Test Replacement

When the only uses of loop parameter values are in a strength reducible context, as occurs when the parameter is used as an array index; the source values for loop bounds are never used except of controlling the loop. By creating a parallel loop, with the loop initial value, incremental value and terminal value modified as required by the strength reduced reference, the original test and assignments may be deleted.

3.3.10.2.4 Loop Collapse

When a loop contents consist solely of its own loop control code and an inner loop, there are often circumstances where the loop control may be combined for the two loop into a single loops. This situation is not unusual when performing matrix operations. The conditions to be satisfied for loop collapse is that the loop parameter values obey the following relationship:

- The repetition count of the inner loop is the same as the increment value of the outer loop.
- The upper bound of the nested loop plus the nested loop's increment value is the initial value of the successive nested loop invocations.

In general, this loop form cannot happen directly in Ada but may occur after the application of strength reduction and test replacement.

3.3.10.3 Outputs

The LOOPER writes the IL file for COGEN.

3.3.11 Code Generator - COGEN

The code generator reads the intermediate language (IL) produced by the optimizer and generates the sequences of machine instructions necessary for the object program to perform the operations specified by the IL.

3.3.11.1 Inputs

IL file - written by the optimizer or target dependent optimizer.

Symbol table - resident - produced or modified by the previous Compiler phases (See Appendix A)

3.3.11.2 Processing

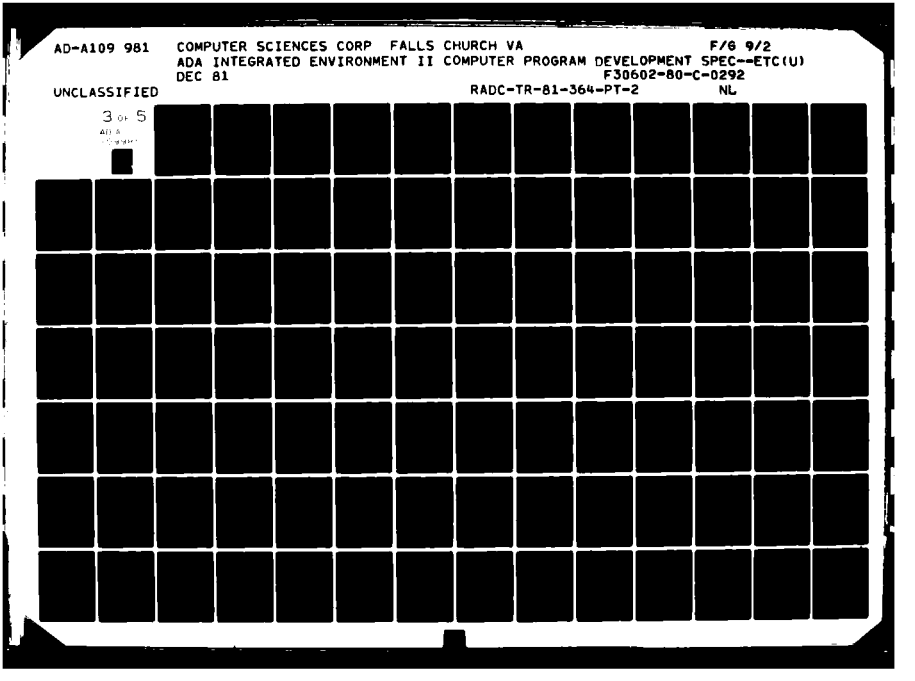
COGEN is responsible for selecting the code sequences necessary to perform, in the target computer, the operations specified by the IL. Although the IL is expressed in reverse Polish notation, code is not generated directly from the IL. Rather, a portion of the IL is read and used to construct an IL tree which forms the basis for selecting code sequences. Thus code can be generated in context.

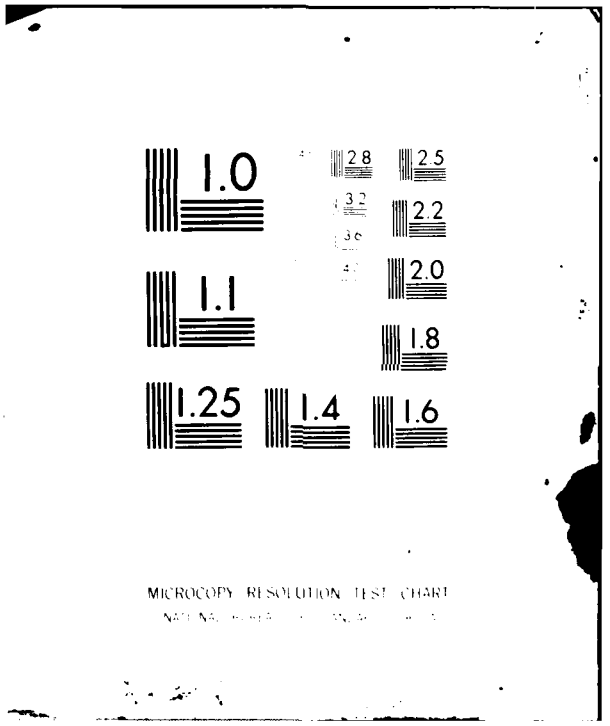
An example of where context information obtainable from a tree representation is needed is in the evaluation of relationals in IF statements. On most machines there is no need to actually compute a boolean value for the relational. It suffices to perform a test that sets the condition codes, followed by a conditional branch.

The code generator is responsible for generating code that is locally good, or can be made locally good by POST. The responsibility for the quality of code on a global basis belongs to the optimizer.

The code generator may knowlingly choose to generate locally suboptimal code if it is easier for POST to rectify the situation than for the code generator to generate the preferable code sequence originally.Such a situation can arise if adjacent instructions are generated from different parts of the IL tree or by different routines in the code generator. One such example is a test instruction that follows an instruction that has already set the condition code correctly. It is certainly easier for COGEN

AD-A109 981 COMPUTER SCIENCES CORP FALLS CHURCH VA F/G 9/2
ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPEC--ETC(U)
DEC 81 F30602-80-C-0292
UNCLASSIFIED RADC-TR-81-364-PT-2 NL




1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

to generate the test unconditionally than to keep track of previous settings of condition codes, and it is not difficult for POST to remove such superfluous tests.

The code generator is probably the most target-dependent phase in the Compiler, because it must have the most intimate knowledge of what instructions are available on the target and what the semantics of these instructions are. Much of the logic in COGEN shall be reusable for code generators for different tragets, however. The creation of the internal form of the IL and the tree-walking that drives the selection of code sequences are largely machine independent. COGEN shall be designed so that the target-dependent and target-independent sections are marked as such, and can be separated due to their modular design.

COGEN is, to a large extent, language-independent. It is driven by the IL and the symbol table, and is concerned with the internal representation of data, rather than with source language operations and Ada types. Thus, the code generator does not address such language issues as overloading and generics. These are handled in the front end.

There are, of course, aspects of code generation that are peculiar to Ada. These include the handling of tasking, exceptions, and some of the attributes such as VALUE and IMAGE.

3.3.11.3 Outputs

Code File

The code file provides the primary means of communication between COGEN and POST and between POST and ASM. It contains sequences of machine instruction tokens, together with other supplemental information that is used in the generation of the object module and the assembly listing.

The code file is, to a large extent, target independent, even though it is used to describe target machine instruction sequences. Much of the supplemental information is valid regardless of target. Some entries are, by their very nature, target-dependent. The prime example of this is the instruction entry.

The following is a definition of the code file tokens in pseudo-Ada. It should be noted that constraints are not specified, nor are incomplete type definitions provided.

```
    type CONDITION is new BOOLEAN;
--turns code on or off

    type CONDITION_REF is access CONDITION;

    type CODE_FILE_TOKEN_ENUM is (COMMENT, CONSTANT, END_COND, END_FILE,
                                  INSTRUCTION, LABEL, LINE, MACRO_CALL_START,
                                  MACRO_CALL_END, MACRO_PARM_OBJECT,
                                  MACRO_PARM_IMMEDIATE, START_COND);

    type CODE_FILE_TOKEN (CLASS: CODE_FILE_TOKEN_ENUM) is
    record case CLASS is
          when COMMENT =>

              COMMENT_STRING : STRING;
--comment is character

         when CONSTANT =>
              CONSTANT_VALUE : TARGET_WORD_TYPE; --target dependent

         when END_COND | START_COND  =>              --conditional code
              CONDITION : CONDITION_REF;

         when END_FILE => null;

         when INSTRUCTION =>
              OP_CODE    : OP_CODE_ENUM;
              INST_FORMAT: INST_FORMAT_ENUM;
              -- target dependent fields

         when LABEL =>                               --label symtab entry
              LABEL_VALUE : LABEL_SYM_REF;

         when LINE =>
              SOURCE_KEY  :  SOURCE_KEY_TYPE;        --source key

         when MACRO_CALL_START | MACRO_CALL_END =>
              MACRO_ID    : MACRO_ID_ENUM;           --macro "name" (target)
```

```
        when MACRO_PARM_OBJECT = >
             OBJECT      : SYM_REF;                --macro actual parameter
        when MACRO_PARM_IMMEDIATE =  >
             IMMEDIATE   : INTEGER;                --macro actual parameter
        end case;
        end record;
```

The uses of the various entries are as follows:

COMMENT - used to pass information through to the assembly listing. A comment might be used, for example, to indicate that the code to follow was part of an exception handler

CONSTANT - used to cause a constant to be generated in the code. One such use would be to generate portions of a procedure's preamble.

START_COND and END_COND - are brackets that are used to delimit sections of conditional code. Code that appears between these brackets is suppressed if the condition pointed to by the brackets is set to false. The conditional code feature allows one phase (say COGEN) to generate code before it is sure whether the code is necessary. Later on, when it is known if the code is needed, a condition flag can be set. The following phase (POST in this example) can include or exclude the code based on the flag's value.

One example of where this feature can be used is in allowing a common subexpression to be copied at the point of its computation in case it is destroyed before all uses have been satisfied. If the copied value is ever needed, COGEN can leave the condition on. If the code is unnecessary, the condition is turned off and the code is deleted by POST when it encounters the START_COND in the code file.

END_FILE - marks the end of the code files.

INSTRUCTIONS - contain the information necessary to format a target machine instruction. This is a target-dependent code file entry, although many of the fields are common to the various targets. The representation shown here is a generalized form.

LABEL tokens - denote label defintions. The two main purposes for such tokens are that POST can take flow of control into account, and that labels can be included in the assembly listing.

LINE tokens - allow the source line to be related to the generated code for that line. This is used both for printing the source key in the assembly listing and to aid in Compiler maintenance.

START_MACRO_CALL - and END_MACRO_CALL are used to delimit the parameters for Code File macros. The only code file entries between the START and END are either of the two MACRO_PARMs described below. POST performs the macro expansion, substituing the actual macro parameters supplied in the code file. The macros available for a given target are determined during Compiler design.

MACRO_PARM_OBJECT - is an actual parameter that is an object whose definition appears in the sumbol table.

MACRO_PARM_IMMEDIATE - is an immediate value which is the actual parameter for a macro call. This code file type exists to avoid cluttering the symbol table with constants that are used in macros.

DOUBLE_ADCON, LEFT_ADCON, and RIGHT_ADCON tokens - are used to cause address constants (adcons) to be output in the code. DOUBLE_ADCON is used for targets such as the DEC-10 that allow addresses in both halves of the machine word. LEFT_ADCON and RIGHT_ADCON are used to generate adcons in the left and right halves of the target word, respectively. Only RIGHT_ADCON is applicable to the IBM 370; the others are included in the interest of retargetability.

3.3.11.4 *Special Requirements*

In order to simplify its task, COGEN may require that POST make certain improvements in the generated code. The improvements are target-dependent.

3.3.12 Post Code Generation Optimizer - POST

The post code generation optimizer which will be called POST) improves the generated code by performing various transformations on the object code that was produced by the code generator. Among these transformation are what are commonly termed "peephole" optimizations, but more global optimizations, namely branch optimizations and constant pooling are also performed.

3.3.12.1 Inputs

Code file - written by code generator.

3.3.12.2 Processing

The processing performed by POST includes code improvements that are both machine-independent and machine-dependent in nature. Since the primary input to this phase is the code file, which contains, of necessity, machine-specific information, this phase will require some modification if it is to be used for a new target machine. However, much of the underlying skeleton should be reusable for different targets.

In addition to these sorts of code improvements, POST serves to make COGEN's job easier in several other ways. Based on flags that the code generator has set, conditional code is included or excluded by POST. (See 3.3.8.3 Code File for a description of conditional code.) POST also expands Code File macros.

POST first reads a portion of the code file into memory. Ideally, it shall be possible to maintain the code for an entire procedure, task or package body in core at one time. POST shall be able to work on as much code as will fit, by reading until an end-of-body is encountered or until a size limit is reached. The only negative effect will be that certain branching optimizations will not be performed if a branch and its destination are not in core simultaneously.

The instructions read in from the code file shall be maintained as a doubly-linked list so that instructions can be inserted or deleted easily (in the hope that the delections will outnumber the insertions). Superimposed on this structure will be branch links that connect branch instructions with their destinations.

Optimizations that are performed by this phase (depending upon the characteristics of the target computer) are:

1. Constant pooling
2. Cross jumping
3. Combining adjacent instructions
4. Branch optimizations
5. Other optimizations

Constant pooling involves finding constants common. Constants shall be pooled on the basis of their binary representations, rather than on the basis of their types and source representations.

Cross jumping involves examining the point where different execution paths merge (i.e., labels). If the instructions on the ends of the two paths are the same, the instructions can be deleted from the end of one sequence, provided that a branch is inserted to transfer control to the corresponding point in the other sequence.

There are many opportunities for combining adjacent instructions on most target machines. A number of machines have instructions which increment or decrement a counter and then branch as a result of the counter reaching a certain value. The code generator may generate separate instructions to modify the operand and to jump. POST can substitute the single more powerful instruction for two or more instructions which, together, perform the same function.

Other examples of combining adjacent instructions include:

1. Deleting loads that immediately follow stores of the same operand.
2. Deleting explicit condition code setting instructions following operations that set the condition codes.

There are various branch optimizations which can be performed. These include changing:

```
Bc          11--Condition1 branch to
                11
B           12--Unconuitioal branch to
                12

11:
```

to:

```
Bnotc 12    --Conditional branch to
                12

            --with complemented

                condition.
```

Other branch optimizations include using short branch instructions where possible. This optimization shall be performed at the end of POST, since other optimizations may reduce the size of the program and may, therefore, affect the number of branches for which the short form may be used.

Having modified the code for a procedure or portion thereof, POST writes out the modified code file.

3.3.12.3 Outputs

Code file (modified) - to be read by the assembler.

3.3.13 Assembler - ASM

The assembler phase of the Compiler reads the code file produced by the post code generation optimizer (POST) and generates a relocatable object module. In addition, the assembler is responsible for producing the assembly listing. It should be noted that although this phase is called the "assembler", it does not read assembly language source, but rather the internal form of machine instructions from the code file. ASM's function corresponds to that of the second pass of a two-pass assembler. ASM's design shall accomodate its use as the basis of such a second pass, should an assembler be required for some target.

3.3.13.1 Inputs

Code file - file written by POST (See 3.3.12.3)

Symbol table -(resident)- produced or modified by previous Compiler phases (See Appendix A).

Name table - file written by front end (See Appendix Z).

Compiler options - in compilation control record (See 3.3.1.3)

3.3.13.2 Processing

The primary responsibility of the assembler (ASM) is to generate the relocatable object from the code file, and to produce a pseudo-assembly listing if one is requested. In addition, the resolution of addresses is performed by ASM.

The resolution of addresses must be performed by the assembler before the code file is processed. ASM walks the symbol table and changes those addresses that are relative to some Compiler internal location counter (e.g., the start of constants) to be relative to the start of a linker control section (CSECT).

The name table must be read back into core so that the symbolic representation of names is available for use in producing the assembly listing and, in a later phase, the cross-reference and attribute listing.

The generation of the relocatable object module from the code file is relatively straightforward. It consists principally of formatting the

instructions based on the information contained in the code file and the symbol table. Since the instructions being formatted are target machine instructions, the assembler is necessarily target-dependent. Much of the logic of this phase, however, is applicable to a variety of targets.

The main loop logic, that is, the reading of the code file and switching on the type of code file entry can be retained despite retargetting. While some fields in the code file are machine-dependent, there is enough commonality between machine architectures so that the basic form of the code file is relatively constant.

Much of the effort required to retarget the assembler can be confined to the data declarations. For example, although op code fields on different targets may be of different sizes and be located at different positions, the judicious use of field names, enuneration types, and table lookups can minimize the changes necessary for retargetting.

Similarly, if a careful distinction ismade between instruction units, addressable units, and word size, retargetting is made easier.

A secondary function of the assembler is to generate the (pseudo-) assembly listing. This listing is described in Appendix C. The assembly listing provides a human-readable representation of the Compiler's principal output. The listing itself shall be in a standard format to minimize retargetting costs.

The generation of assembly source is not proposed in the design. If it becomes necessary for some reason to generate assembly source, the assembler phase would be the logical place to do it. Note that the pseudo-assembler source portion of the assembly listing is not actually able to be input to an assembler due to the fact that names appearing in different scopes would be multiply defined.

3.3.13.3 Ouputs

Relocatable objects

Pseudo-assembly listing

3.3.14 Cross-reference Generator - XREF

The cross-reference listing generator produces the cross-reference and attribute listing and generates the debugging tables for the object module.

3.3.14.2 Inputs

Cross-reference file - (See 3.3.4.3)

Symbol table - resident - (See Appendix A)

Name table - resident - (See Appendix A)

Compiler options

3.3.14.2 Processing

The processing performed by XREF is relatively straightforward. The cross-reference file is read into core and an in-core sort is performed. The sorting method used will be Hoare's Quicksort. In order to guard against worst-case behavior, the median of three elements shall be used to choose partitioning elements.

The sort key(s) used depend(s) on which listing option is selected. For the straight alphanumeric listing, the major key is the name and the minor key is the line number of the reference. For the structured listing, the major key is the outermost structure, wth inner structures as minor keys. Alternatively, the sort could be thought of as using fully qualified names, sorted component by component.

After the sorts have been performed, the listing is printed. Set/use information is obtained from the sorted cross-reference tokens. Attribute information is obtained from the symbol table.

In parallel with the listing generation, XREF writes set/use information for package data and external routines to the object file, so that the file can be used later by the linker to generate a system cross-reference.

The other major responsibility of XREF is to generate the debugging tables, which are then written to the object module. This is primarily an exercise in formatting the symbol table in such a way that it is useful to the debugger.

3.3.14.3 Outputs

Cross-reference listing - to listing file - (See Appendix C)

Attribute listing - to listing file - (See Appendix C)

Concordance information - to object file

Debug tables - to object file

3.3.14.4 Special Considerations

XREF requires sufficient table space to perform a memory-resident sort of all of the cross-reference tokens generated by the compilation of a particular compilation unit. If insufficient space is available, the cross-reference listing shall be partitioned.

3.3.15 Library Unit Specification Update - LIBO

LIBO runs after the semantic analysis phase and produces the Library unit specification of the program being compiled.

3.3.15.1 Inputs

The inputs to LIBO are the resident symbol table and optionally the generic declaration file.

3.3.15.2 Processing

LIBO formats generic skeletons and inline subprogram bodies for inclusion into the Program Library. The current compilation-unit name with a date/time stamp will be added to all withed Library units. If this compilation declared any separate subunits, the symbol table is copied into the Program Library -- the part that is required for subunit compilation -- and creates a null body stub entry in the Program Library. If this null entry is not replaced before linking, the Linker will substitute a canned procedure body for the reference.

LIBO will check if this compilation-unit's spec already exists in the Library and if the visible interfaces differ. If so, LIBO produces a list of probable obsolete compilations.

200

This compilation-unit's spec is added to the Library and relocatable object. The previous version unit spec will be deleted from the Library. Finally, LIBO produces the ADTs.

3.3.15.3 Outputs

The output of LIBO is the Library unit specification of the program being compiled, the ADTs and the list of possible recompilation needs.

3.4 ADAPTATION

This section describes the requirements of the Compiler with respect to system environment, system parameters, and system capacities.

3.4.1 General Environment

Not Applicable.

3.4.2 System Parameters

Not applicable.

3.4.3 System Capacities

The size of the memory partition allocated to the Compiler will affect its performance - the Compiler will be organized to page unit data into limited space, and to take advantage of dynamic memory allocation, when available.

3.5 CAPACITY

Not applicable.

SECTION 4 - QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the Compiler. The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the Compiler. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data are processed. For example, a review of hard copy test data might be used to verify that the values of specific parameters are correctly computed.

202

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

4.1.1 Subprogram Testing

Following unit testing, individual modules of the Compiler shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2. Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

CPCI testing shall be performed on all development software of the Compiler. This specification presents the performance criteria which the developed CPCI must satisfy. The correct performance of the Compiler will be verified by testing its major functions. Successful completion of the program testing that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3. System Integration Testing

System integration testing involves verification of the integration of the Compiler with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the Compiler performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

200

SECTION	TITLE	INSP.	SPEC.	DEMO.	DATA.	PARA. NO.
3.3.1	Compiler Exec		X			4.2.3
3.3.2	Lexical Analysis		X			4.2.3
3.3.3	Library Specification Input		X		X	4.2.2,4.2.3
3.3.4	Res. & Sem. Analysis	X			X	4.2.1,4.2.2
3.3.5	Allocator		X		X	4.2.2,4.2.3
3.3.6	Checks		X		X	4.2.2,4.2.3
3.3.7	Target Optimizer		X		X	4.2.2,4.2.3
3.3.8	Flow Analyzer		X		X	4.2.2,4.2.3
3.3.9	Optimizer		X		X	4.2.2,4.2.3
3.3.10	Loop Processor		X		X	4.2.2,4.2.3
3.3.11	Code Generator	X	X			4.2.1,4.2.3
3.3.12	Post Code Generation Optimizer		X			4.2.2,4.2.3
3.3.13	Assembler	X	X			4.2.1,4.2.3
3.3.14	Cross Reference Gen.		X		X	4.2.2,4.2.3
3.3.15	Lib.Unit Spec Update					4.2.2,4.2.3

Table 4-1 Test Requirements Matrix

4.2.1 Inspection

Output listings shall be inspected to ensure that they match relevant documentation.

4.2.2 Review of test data

Drivers shall be written to generate input data and to log output data. Test input scripts and expected test output shall be developed in accordance with specifications. Testing shall consist of comparing expected output data with test output data.

4.2.3 Special Tests

Each function is tested as a part of the compilation process.

4.3. ACCEPTANCE TESTING

The Compiler shall be submitted for acceptance testing by the Government using the Ada Compiler Validation Facility. Satisfactory performance of the Compiler in this testing shall result in the final delivery and acceptance of the Compiler.

266

SECTION 5 - DOCUMENTATION

5.1 General

The documents that shall be produced during the implementation phase in association with the Ada compiler development are:

1. Computer Program Development Specification
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. Users Manual
6. Retargetability/Rehostability manual
7. Language Reference Handbook

5.1.1 Computer Program Development Specification

The final Ada Compiler B5 Specification shall be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II. A single document shall be prepared for the compiler that defines the functional capabilities and interfaces. Any dependencies on the host and target shall be addressed in the document. Additionally, characteristics of potential hosts and targets that have had impact on the B5 specification shall be presented.

5.1.2 Computer Program Product Specification

A type C5 specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document shall be used to specify the compiler design and development approach for implementing the B5 specification. This document shall provide the detailed description that shall be used as the baseline for any Engineering Change Proposals. A single C5 shall be produced for the compiler with different sections addressing the dependencies of the two host computers.

5.1.3 Computer Program Listings

Listings shall be delivered that are the result of the final compilation of the accepted compiler. Each compilation unit listing shall contain the corresponding source, cross-reference and compilation summary. The source listing shall contain the source lines from any INCLUDEd source objects.

5.1.4 Maintenance Manual

A Compiler Maintenance Manual shall be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the compiler to be easily maintained by personnel other than the developers. The documentation shall be structured to relate quickly to program source. The procedures required for debugging and correcting the compiler shall be described and illustrated. Sample run streams for compiling compiler components, for relinking the compiler in parts or as a whole, and for installing new releases shall be supplied. The data base shall be fully documented with pictures of record layouts where appropriate and data algorithms explained.

The Maintenance Manual shall be organized with a standard outline and separate parallel volumes shall be delivered that address the tailoring of the compiler to a particular target or host computer. Debugging aids that have been incorporated as an integral part of the compiler shall be described and their use fully illustrated. Special attention shall be given to the description of the maintenance mode operation of the compiler used to aid in the pinpointing of compiler problems.

5.1.5 Users Manual

A Users Manual shall be prepared in accordance with DI-M-30421 and shall contain all information necessary for the operation of the compiler. Because of the virtual user interface presented by the ACLI, a single manual is sufficient for all host computers. Separate appendices shall describe such machine dependent data as the target machine parameters, procedure calling and register conventions, representation attributes, packages STANDARD and SYSTEM, low-level I/O, machine code insertions and the INTERFACE pragmas. Sample compiler listings shall be included in the manual.

A complete list of all compiler diagnostic messages shall be included with supplemental information chosen to assist the programmer in locating and correcting source program errors.

5.1.6 Retargetability/Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual shall be prepared that describes step by step the procedures for retargeting the Ada Compiler to a different computer, integrating this new "back-end" into the compiler system, and transporting the compiler onto different host computers. Tips shall be provided that will guide the developer module by module as to what may be used unchanged entirely or in part. In particular, the adaptation of the machine dependent optimizer shall be carefully presented to permit maximum benefit to be derived from the remaining machine-independent optimization phases.

5.1.7 Language Reference Handbook

An Ada Language Reference Handbook shall be prepared that shall contain syntax diagrams for all language constructs and a cross-reference listing of all constructs. Other handy information such as number conversion and ASCII encoding tables, host and target parameters, and a compiler option summary shall be included.

APPENDIX A - SYMBOL TABLE

This appendix describes the symbol table and the three closely related tables: the hash table, the homonym table and the name table. The symbol table contains attribute information for objects defined in Ada programs, as well as for compiler-defined objects. The other three tables provide a means of associating the attributes of an object with its source name.

The symbol table described here is patterned after the symbol table portion of TCOL. It has been updated to reflect the language changes between preliminary and revised Ada, and to allow the symbol table to serve all phases of the compiler. ($TCOL_{Ada}$ as documented, is an interface between the front and back-ends of the compiler, and does not, therefore, include fields that may be needed by one but not the other.)

The symbol table shall be resident for the entire compilation. The name, homonym and hash tables shall only be resident for a portion of the compilation. The hash and homonym tables are only needed in order to map from a character string (name) to a symbol table entry. These tables need be resident only until the last package specification has been read. In particular, they shall not be resident during the optimization, code generation, and editor phases.

The name table is required both for mapping from source names to symbol table entries and for producing listings. Thus, although the name table is not resident during optimization and code generation, it must be written out so that it can be used during the generation of diagnostics and the cross-reference, attribute and assembly listings.

The hash table provides a mapping from a hash value (obtained by applying a hash function to a name string) to the name pointer array. The hash table also contains lists of pointers to names that hashed to the same value. (Chaining is used to resolve collisions).

The name table is divided into two parts: the name pointer array and the name table proper. The name pointer array contains pointers both to the names themselves, and to a list of homonyms (symbol table entries associated

The structure of these tables is depicted below:

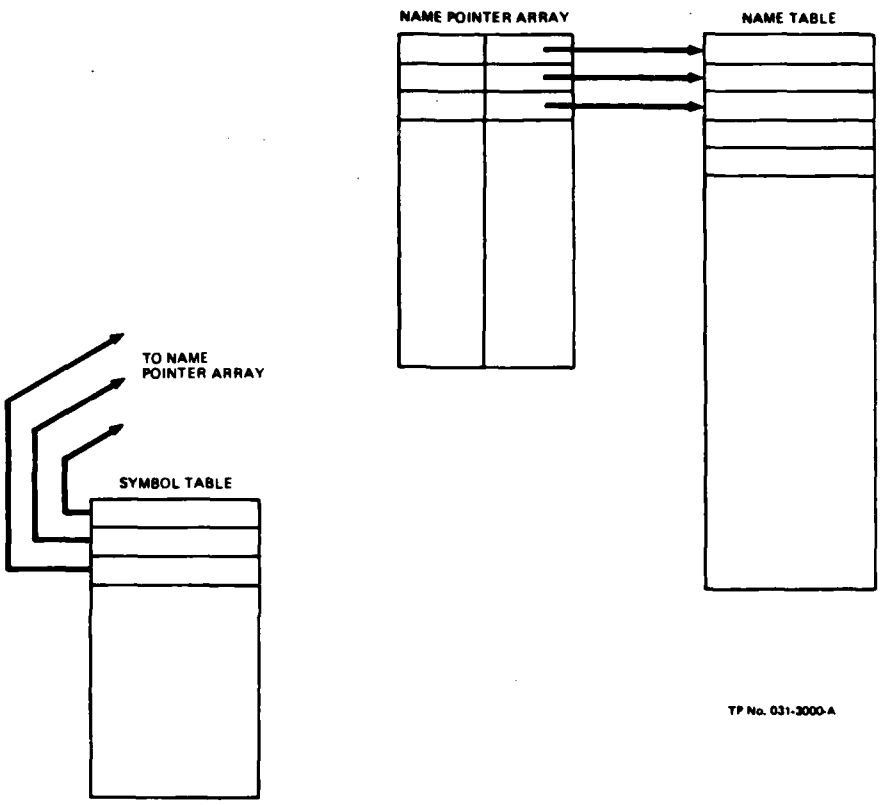


Figure A-1. Name Table Structure (In LEX)

The following picture illustrates the symbol table and name table structure as seen in the assembler phase:

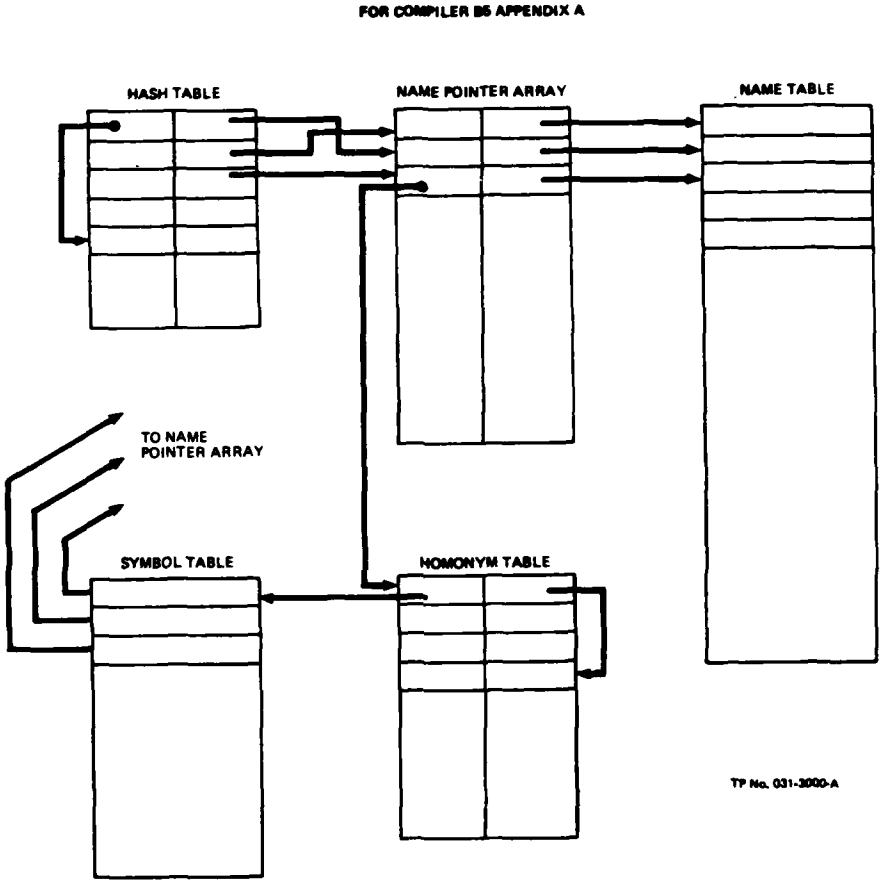


Figure A-2. Name Table Structure (In ASM)

212

with the same name). The name pointer array exists for two reasons. First, it allows name numbers to be stored in the symbol table. If the name table is written out in order, a new name pointer array can be constructed when the name table is read back in, and the name associated with a given symbol table entry can be found. Second, since the homonym table is only needed in the front end, the homonym list pointers in the name pointer array do not have to be rebuilt when the names are read back in.

The name table contains, for each unique name, a list of symbol table entries that define objects (or procedures, etc.) which possess that name. This table is used during name resolution and is not needed by the compiler back-end.

Following is a description of the hash table, the name table, the homonym table and the symbol table. A certain amount of license has been taken with the descriptions. These liberties include:

- Incomplete type definitions are omitted.
- Declarations are grouped according to table, rather than in the order required for elaboration.
- Constraints on the arrays are omitted rather than chosen arbitrarily.

```
Hash Table
type HASH_INDEX is new INTEGER;              --bounds for hash table
type HASH_TABLE is
    record
    NAME_NO: array(HASH_INDEX) of NAME_INDEX;  -- points into name
                                                  pointer
    LINK:    array(HASH_INDEX) of HASH_INDEX;  -- link for
                                                  collisions
    end record;
HASH:  HASH_TABLE;                           -- hash table object
```

```
Name table - names are allocated on the heap

type NAME is new STRING;
type NAME_PTR is access NAME;                    -- name pointers

Name pointer array (for front end)

type NAME_INDEX is new INTEGER;                  --bounds for name
                                                 --pointer array

type NAME_POINTER is
    record
    NAME_REF:     NAME_PTR;                      -- pointer to name
    HOMONYM_HEAD: HOMONYM_REF;                   -- pointer to homonym
                                                     list head

    end record;

NP: array(NAME_INDEX) of NAME_POINTER;           --name pointer array

Homonym table
Links symbol table entries that have the same name

type HOMONYM is
    record
    DEFN: SYM_REF;                               --points to symbol
                                                 --table
    NEXT: HOMONYM_REF;                           --points to next
                                                 --homonym

    end record;
type HOMONYM_REF is access HOMONYM;


Enumeration types for symbol table

type SYM_CLASS_ENUM is (BLOCK_SYM, CASE_SYM, CHOICE_SYM,
                        ENTRY_SYM, ENUMERAL_SYM, EXCEPTION_SYM,
                        GENERIC_INST_SYM, GENERIC_SYM, LABEL_SYM,
                        LITERAL_SYM, PACKAGE_SYM, PRAGMA_SYM, RANGE_SYM,
                        SELECT_ALTERNATIVE_SYM, SELECT_SYM, SUBPROGRAM_SYM,
                        SUBTYPE_SYM, TASK_SYM, TASK_TYPE_SYM, TYPE_SYM,
                        VARBL_SYM, VARIANT_CASE_SYM, VARIANT_SYM);
```

```
type ALTERNATIVE_ENUM is (DELAY, TERMINATE, ACCEPT);
type CONSTANCY_ENUM is (NOT_CONSTANT, UNKNOWN, COMPILE_TIME,
                        LINK_TIME, EXECUTION_TIME);
type CHOICE_KIND_ENUM is (RANGE, INDIVIDUAL, OTHERS);
type LINKAGE_ENUM is (ADA, SYSTEM,others);
type LIT_KIND_ENUM is (INT_LIT, ENUM_LIT, FLOAT_LIT, FIXED_LIT,
                        STRING_LIT, EXPR_LIT);
type NUMERIC_REP_ENUM is (FIXED_NUMERIC, FLOAT_NUMERIC,
                          INTEGER_NUMERIC, DISCRETE,
                          GENERIC_INTEGER, GENERIC_FIXED,
                          GENERIC_FLOAT);
type PEDIGREE_ENUM is (DECLARED, DERIVED, REDEFINED);
type REP_ENUM is (ACCESS_REP, ARRAY_REP, ENUMERATION_REP,
                  NUMERIC_REP, RECORD_REP);
type SELECT_ENUM is (ACCEPT, COND_ENTRY, TIMED_ENTRY);
type SPECIES_ENUM is (VARBL, IN_PARM, OUT_PARM, IN_OUT_PARM,
                        RECORD_COMPONENT);
type SUB_DEFAULT_ENUM is (NONE, AT_INSTANTIATION, DECLARED);
type SUB_KIND_ENUM is (PROCEDURE, FUNCTION);

Other types

type REL_ENUM is (ABS,REL,EXT);
type LOCATION is INTEGER;
type LOCATION_SPECIFIER is
    record
        RELOCATION:REL_ENUM;
        CSECT:INTEGER;
        SECTION_OFFSET:INTEGER;
    end record;

References (access types);
(incomplete declarations are omitted
full definitions follow symbol table
```

```
Symbol table proper

type SYM(CLASS : SYM_CLASS_ENUM) is        -- Symbol table
       record
           case CLASS is

               when ENTRY_SYM | EXCEPTION_SYM | GENERIC_INST_SYM |
                   LABEL_SYM | PACKAGE_SYM | PRAGMA_SYM |
                   SUBPROGRAM_SYM | SUBTYPE_SYM | TASK_SYM |
                   TASK_TYPE_SYM | TYPE_SYM | VARBL_SYM =>
                 NAME                      : NAME_REF;
                 case CLASS is

                   when ENTRY_SYM =>
                     ENTRY_FORMALS         : VARBL_SYM_SEQ;  -- formal
                                           --parms for entry
                     ACCEPTS               : access ACCEPT_LIST;  --list of
                                           --accepts
                     ENTRY_RANGE           : RANGE_CONSTRAINT_REF;
                                           --range for entry
                                           --family
                     ENTRY_LOCATION        : LOCATION;
                                           -- from for use at
                     ENTRY_NUMBER          : INTEGER;
                                           -- ordinal number of

                   when EXCEPTION_SYM =>
                     null;

                   when GENERIC_INST_SYM =>
                     INSTANCE_OF           : GENERIC_SYM_REF;
                                           -- what was
                                           -- instantiated
                     GENERIC_ACTUALS : GENERIC_ACTUAL_SEQ;
                                           -- actual parameters
```

```
when LABEL_SYM =>
  LABEL_LOC              : LOCATION_SPECIFIER;
                         --defn of label
  FORWARD_TARGET         : BOOLEAN;
                         -- if target of fwd br
  FORGET                 : BOOLEAN;
                         -- don't remember regi
  REFERENCE_COUNT        : INTEGER;
                         -- label references

when PACKAGE_SYM =>
  IS_BUILT_IN            : BOOLEAN;
                         -- for built-ins
  IS_SEPARATE            : BOOLEAN;
  MODULE_LOCATION        : LOCATION;
                         --from for use at
  BODY                   : BLOCK_SYM_REF;
                         -- body entry in symbol

  SPEC                   : -- to be defined

when TASK_TYPE_SYM =>
  IS_SEPARATE            : BOOLEAN;
  MODULE_LOCATION        : LOCATION;
                         --from for use at
  BODY                   : BLOCK_SYM_REF;
                         -- body entry in symbol
  ACTIVATION_LENGTH      : VARBL_SYM_REF;
                         -- temp for alength
  MAX_ACCEPT_NESTING     : INTEGER;
                         -- rendezvous nesting
  NUMBER_OF_ENTRIES      : INTEGER;
                         -- counting families
  ENTRIES                : ENTRY_SYM_SEQ;
                         -- entries for this
```

```
when SUBPROGRAM_SYM =>
  IS_BUILT_IN            : BOOLEAN;
                         -- for built-ins
  IS_SEPARATE            : BOOLEAN;
  MODULE_LOCATION        : LOCATION;
                         --from for use at
  BODY                   : BLOCK_SYM_REF;
                         -- body entry in symbol
  SPEC                   : -- to be defined
  SUBPROGRAM_KIND        : SUB_KIND_ENUM;
                         --(PROCEDURE | FUNCTION)
  SUBPROGRAM_FORMALS     : VARBL_SYM_SEQ;
                         -- formal parameters
  LINKAGE                : LINKAGE_ENUM;
                         -- (ADA | SYSTEM)
  MACHINE_CODE_IN_PROC   : BOOLEAN;
                         -- machine insertions
  GLOBAL_NAMES_LIST      : GNL_REF;
                         -- for use by optimizer
  RESULT_SUBTYPE         : VARBL_SYM_REF;
                         -- type of value return
  REDUCIBLE              : BOOLEAN;
                         -- can find calls comm

when PRAGMA_SYM =>
  null;

when SUBTYPE_SYM =>
  PARENT_TYPE            : TYPE_SYM_REF;
  PARENT_SUBTYPE         : SUBTYPE_SYM_REF;
  CONSTRAINTS            : CONSTRAINT_SEQ;
```

```
when TASK_SYM =>
  TYPE_OF_TASK          : TASK_TYPE_REF;
                        -- always present
  IS_TASK_ACCESS        : BOOLEAN;
                        -- for access types
  TASK_LOCATION         : LOCATION_SPECIFIER;
                        -- TCB location
                        -- type has code loc

when TYPE_SYM =>
  PEDIGREE              : PEDIGREE_ENUM;
                        -- (DECL | DERIVED)
  DERIVED_FROM          : SUBTYPE_SYM_REF;
                        -- parent
  REPRESENTATION        : REP_REF;
                        -- representation
  IS_PRIVATE            : BOOLEAN;
                        -- private type
  IS_LIMITED            : BOOLEAN;
                        -- only if private
  IS_PACKED             : BOOLEAN;
                        -- pragma pack
  IS_CONTROLLED         : BOOLEAN;
                        -- pragma controlled
  LENGTH                : INTEGER;
                        -- size (target dependent)
  HAS_REF               : BOOLEAN;

when VARBL_SYM =>
  VARBL_SUBTYPE         : SUBTYPE_SYM_REF;
                        -- never points to TYPE
  CONSTANCY             : CONSTANCY_ENUM;
                        -- when value is constant
  SPECIES               : SPECIES_ENUM;
```

```
    INITIALIZE              : LITERAL_SYM_REF;
                            -- initial value, if
    IS_INITIALIZED          : BOOLEAN;
                            -- even if value is
    IS_PRIVATE_VAR          : BOOLEAN;
    POSITION                :-- to be defined
                            -- for user allocated
    VARBL_LOCN              : LOCATION_SPECIFIER;
    REV_COUNT               : INTEGER;
                            -- reference count
    FOLDING_VALUE           : INTEGER;
                            -- value # for folding
    IS_FROM_PACKAGE         : BOOLEAN;
                            -- so loc spec won't
    SCOPE                   : SUBPROGRAM_SYM_REF;
                            -- defining scope
    SIBLING                 : SYM_REF;
                            -- sibling
    PARENT                  : SYM_REF;
                            -- parent structure
end case;

when BLOCK_SYM =>
    DECLARATIONS            : LABEL_SYM_REF;
                            -- label for decl par
    CODE                    : LABEL_SYM_REF;
                            -- label for start of
    EXCEPTIONS              : EXCEPTION_SEQ;
                            -- exceptions handled
    EXCEPTION_LOCATION      : LABEL_SYM_REF;
                            -- label for 1st hand
```

```
when CASE_SYM =>
  EXPRESSION_TYPE        : SUBTYPE_SYM_REF;
                         -- type of selector
  CASE_CHOICES           : CASE_CHOICE_SEQ;
                         -- choices specified

when CHOICE_SYM =>
  CHOICE_KIND            : CHOICE_KIND_ENUM;
                         -- (RANGE | INDIV )
  CHOICE_RANGE           : RANGE_SYM_REF;
                         -- for discrete range
  CHOICE_VALUE           : LITERAL_SYM_REF;
                         -- for single choice

when ENUMERAL_SYM =>
  ENUMERAL_NAME          : LITERAL_SYM_REF;
                         -- name of enumeral
  ENUMERAL_VALUE         : INTEGER;
                         -- representation

when GENERIC_PARM_SYM =>
  PARM                   : SYM_REF;
                         -- ref for actual parm
  SUBPROGRAM_DEFAULT     : SUBPROGRAM_SYM_REF;
                         -- default, if any

  SUBPROGRAM_DEFAULT_KIND : SUB_DEFAULT_ENUM;
                         -- (NONE | INST )

when GENERIC_SYM =>
  GENERIC_FORMALS        : GENERIC_PARM_SYM_SEQ;
                         -- formal parms
  PATTERN                :-- to be defined
                         -- so body can be ret
```

```
when LITERAL_SYM =>
  LIT_SUBTYPE           : SUBTYPE_SYM_REF;
                        -- literal type
  LIT_KIND              : LIT_KIND_ENUM;
                        -- (INT | FLOAT, etc)
  LIT_NAME              : STRING;
                        -- printable
  LIT_VALUE             : VECTOR;
                        -- canonical form
when SELECT_ALTERNATIVE_SYM =>
  ALTERNATIVE_KIND      : ALTERNATIVE_ENUM;
                        --(DELAY|TERM|ACCEPT)
  ALTERNATIVE_LABEL     : LABEL_SYM_REF;
                        -- where to execute
  HAS_WHEN_CLAUSE       : BOOLEAN;
                        -- optimize if uncond
  ACCEPT_ENTRY          : ENTRY_SYM_REF;
                        -- entry being accept

when SELECT_SYM =>
  SELECT_KIND           : SELECT_ENUM;
                        -- (ACCEPT | COND )
  NUMBER_OF_CHOICES     : INTEGER;

  SELECT_ALTERNATIVES   : SELECT_ALTERNATIVE_SEQ;
                        -- alternatives
  HAS_DELAY             : BOOLEAN;
                        -- if delay(s) present
  HAS_TERMINATE         : BOOLEAN;
                        -- if terminate alter
  HAS_ACCEPT            : BOOLEAN;
                        -- if accept alternate
```

```
            when VARIANT_CASE_SYM =>
              DEPENDS_ON            : VARIABL_SYM_REF;
                                    --discriminant
              VARIANTS              : VARIANT_SYM_SEQ;
                                    -- when's

            when VARIANT_SYM =>
              VARIANT_CONDITIONS    : CHOICE_SYM_REF;
                                    -- conditions
              DEFINITION            : VARIABLE_SYM_REF;
                                    -- variant definition

            end case;

      end record;
```

References (access types) for SYM's are

```
type CHOICE_SYM_REF     is access SYM(CHOICE_SYM);
type BLOCK_SYM_REF      is access SYM(BLOCK_SYM);
type ENTRY_SYM_REF      is access SYM(ENTRY_SYM);
type GENERIC_SYM_REF    is access SYM(GENERIC_SYM);
type GNL_REF            is access FLOW_LIST_PTR;
type LABEL_SYM_REF      is access SYM(LABEL_SYM);
type LITERAL_SYM_REF    is access SYM(LITERAL_SYM);
type NAME_REF           is access STRING ;
type RANGE_SYM_REF      is access SYM(RANGE_SYM);
type SUBPROGRAM_SYM_REF is access SYM(SUBPROGRAM_SYM);
type SUBTYPE_SYM_REF    is access SYM(SUBTYPE_SYM);
type SYM_REF            is access SYM;
type TASK_TYPE_REF      is access SYM(TASK_TYPE_SYM);
type TYPE_SYM_REF       is access SYM(TYPE_SYM);
type VARBL_SYM_REF      is access SYM(VARBL_SYM);
```

Sequences
No representation is given. The SEQ's are sequences of access variables that could be implemented as an array of access variables or as a linked list. The particular implementation shall be decided on as part of the second phase. The names used here indicate the type of objects that would be pointed to by the access variables. Name_SEQ represents a sequence of type "name". For example, VARBL_SYM_SEQ denotes a sequence of VARBL_SYM's.

```
type CASE_CHOICE_SEQ is

type CHOICE_SYM_SEQ is

type ENTRY_SYM_SEQ is

type ENUMERAL_SYM_SEQ is

type EXCEPTION_SEQ is

type GENERIC_ACTUAL_SEQ is

type GENERIC_PARM_SYM_SEQ is

type SELECT_ALTERNATIVE_SEQ is

type SUBTYPE_SYM_SEQ is

type VARIANT_SYM_SEQ is

type VARBL_SYM_SEQ is

type REP (REP_KIND: REP_ENUM) is
    record
    case REP_KIND is
      when ACCESS_REP =>
         ACCESS_OF      : SUBTYPE_SYM_REF;  -- subtype pointed to
         IS_CONTROLLED  : BOOLEAN;          -- pragma controlled


      when ARRAY_REP =>
         COMPONENT      : SUBTYPE_SYM_REF;  -- array of
         INDICES        : SUBTYPE_SYM_SEQ;  -- bounds are constr
         MUST_CONSTRAIN : BOOLEAN;          -- whether must constr


      when ENUMERATION_REP =>
         ENUMERATION_LITERALS : ENUMERAL_SYM_SEQ;


      when NUMERIC_REP =>                    -- (could be moved to
         NUMERIC_REP_KIND : NUMERIC_REP_ENUM;  -- (INT | FLOAT, etc.)
```

```
      when RECORD_REP =>
         FIELDS        : SYM_REF;          -- (VARIANT or VARBL)
         DISCRIMINANTS : VARBL_SYM_SEQ;    -- discriminants
         ALIGNMENT     : INTEGER;          -- aligned on 0 mod r
    end case;
    end record;

References (access variables) for REP's

type REP_REF is access REP;

Constraints

type CONSTRAINT (CONSTRAINT_KIND : CONSTRAINT_ENUM) is
    record
    case CONSTRAINT_KIND is
      when DISCRIMINANT_CONSTRAINT =>
         DISCRIMINANT  : VARBL_SYM_REF;     -- discriminant
         DISCRIMINANT_IS_STATIC : BOOLEAN;  -- compile-time constra
         DISCRIMINANT_VALUE : SYM_REF;      -- (LITERAL or VARBL)

      when FIXED_ACCURACY_CONSTRAINT =>
         DELTA_VALUE : LITERAL_SYM_REF;     -- error bound

      when FLOAT_ACCURACY_CONSTRAINT =>
         DIGITS_VALUE : INTEGER;            -- digits of precision

      when INDEX_CONSTRAINT =>
         INDEX_TYPE  : SUBTYPE_SYM_REF;      -- type of index
         INDEX_RANGE : RANGE_CONSTRAINT_REF; -- bounds
```

```
      when RANGE_CONSTRAINT =>
         LOWER_IS_STATIC : BOOLEAN ;        -- compile-time cons
         LOWER_VALUE     : SYM_REF;         -- LITERAL or VARBL
         UPPER_IS_STATIC : BOOLEAN;         -- compile-time cons
         UPPER_VALUE     : SYM_REF;         -- LITERAL or VARBL
   end case;
   end record;
```

References (access variables) for CONSTRAINT's

```
type RANGE_CONSTRAINT_REF is access CONSTRAINT(RANGE_CONSTRAINT);
```

The following five figures illustrate the symbol table structure for Types, Records, Procedures, Tasks and Enumeration types.

SYMBOL TABLE — TYPE STRUCTURE

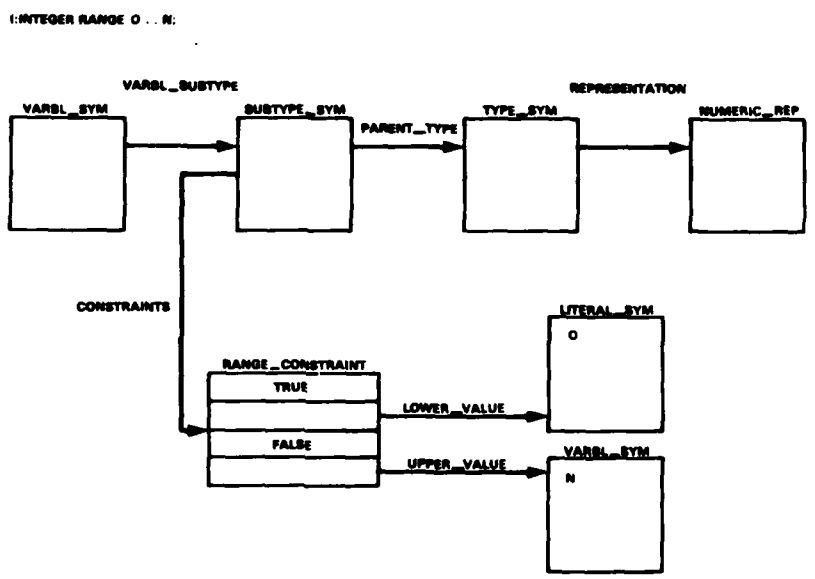


Figure A-3. Symbol Table - Type Structure

227

SYMBOL TABLE — RECORD (WITH VARIANTS)

```
TYPE REC (D: INTEGER RANGE 0.. 3) IS
    RECORD
    A: INTEGER.
    CASE D IS
    WHEN 0 => X,Y:INTEGER.
    WHEN1.. 3 => Z:INTEGER:
    END CASE.
    END RECORD.
R:REC
```

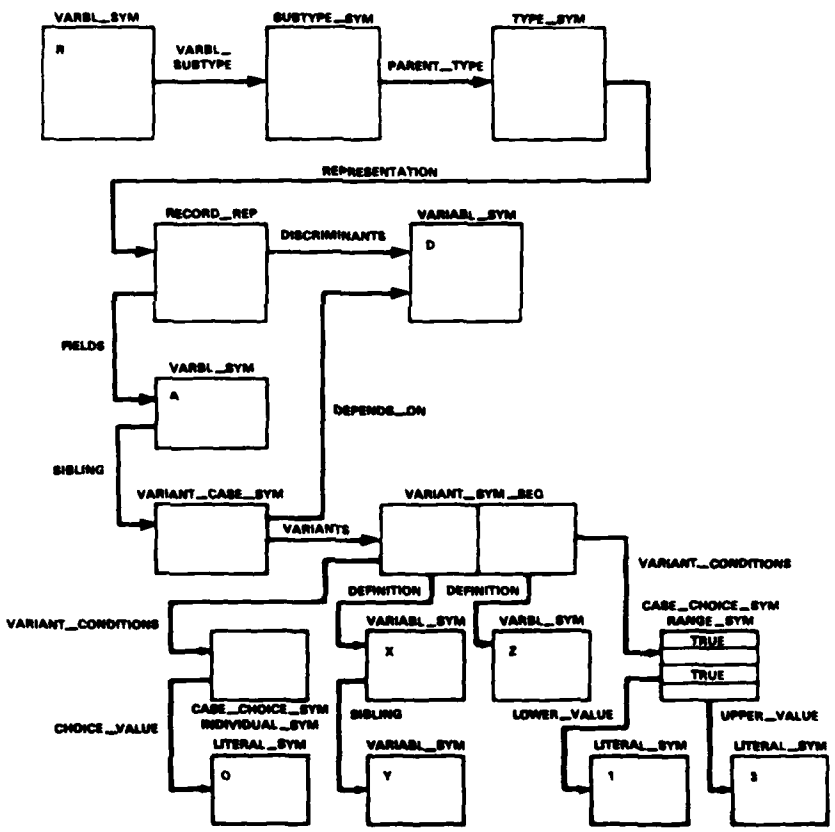


Figure A-4 Symbol Table - Record Structure (variants)

228

SYMBOL TABLE — PROCEDURE

```
PROCEDURE P (I,J:INTEGER) IS
   K,L:INTEGER;
BEGIN
   . . .
END P;
```

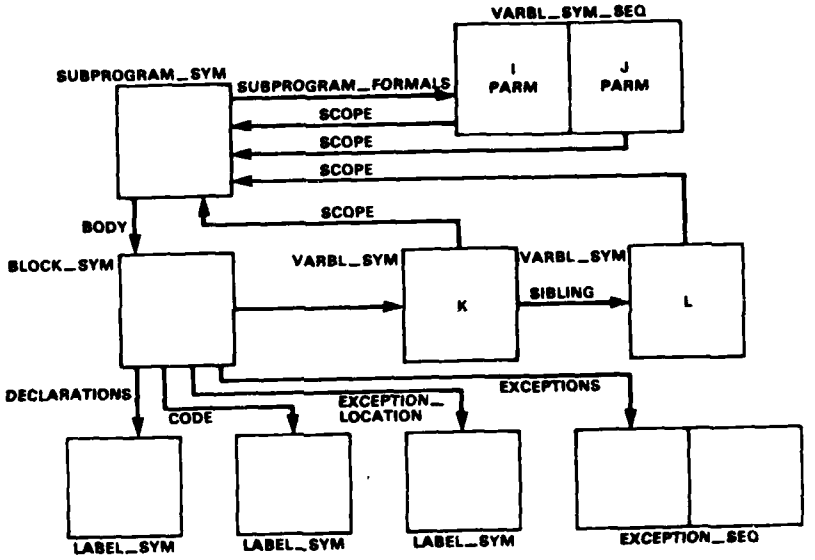


Figure A-5. Symbol Table - Procedure Structure

SYMBOL TABLE – TASK

```
TASK TYPE TT IS
   ENTRY A (I:INTEGER);
   ENTRY B;
END TT;
TYPE TTA IS ACCESS TT;
T:TTA;
```

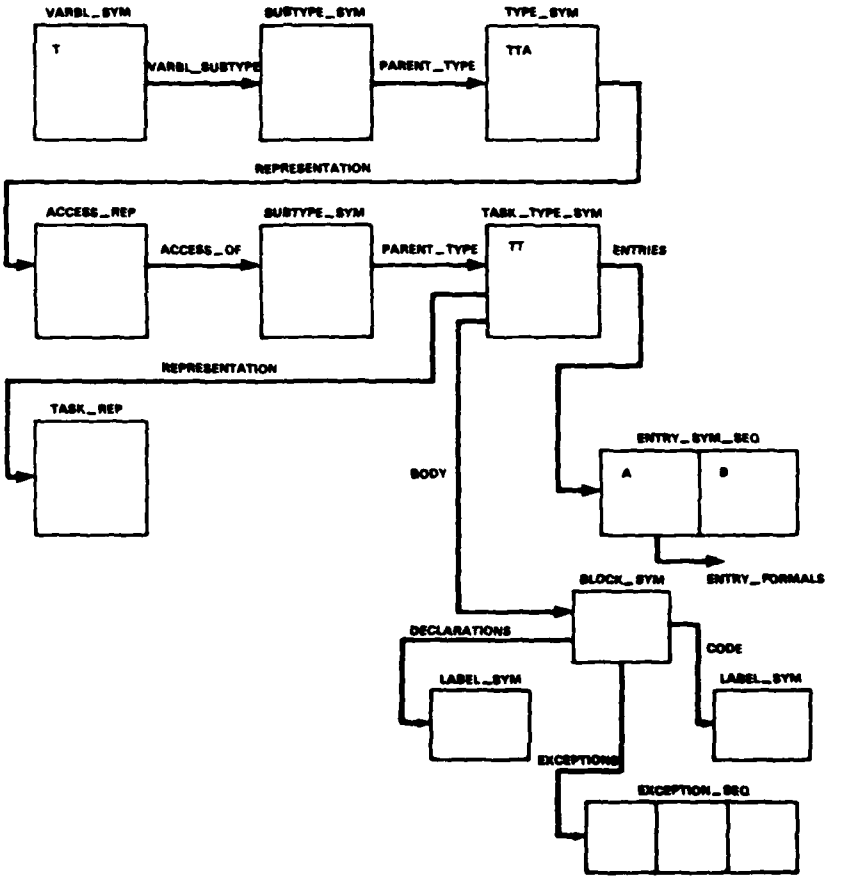


Figure A-6. Symbol Table - Task Structure

230

SYMBOL TABLE – ENUMERATION TYPE

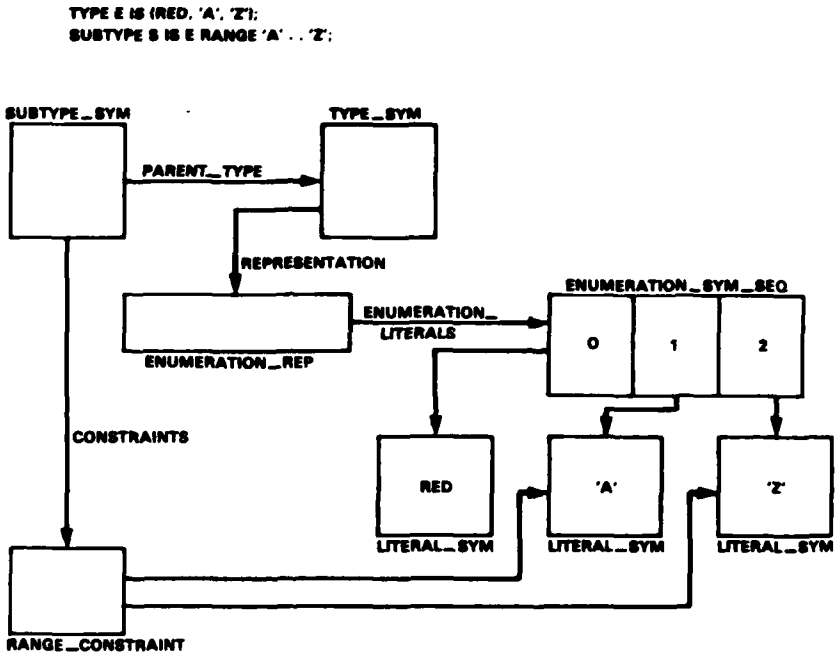


Figure A-7. Symbol Table - Enumeration Type Structure

APPENDIX B - INTERMEDIATE LANGUAGE (IL)

The IL file is an internal representation of the source program. The IL is first produced by RESANL, it is used and modified by the optimizer and is finally input to the code generator phase, COGEN.

An outline of the IL operators is first shown below followed by an Ada description of the IL record structure.

IL OP	Description	Number & type of operands
START_PACKAGE		1 (PACKAGE)
END_PACKAGE		1 (PACKAGE)
START_PROC	START PROCEDURE	1 (PROC)
END_PROC	END PROCEDURE	1 (PROC)
START_TASK	START TASK	1 (TASK)
END_TASK	END TASK	1 (TASK)
START_BLOCK	START BLOCK	1 (BLOCK)
END_BLOCK	END BLOCK	1 (BLOCK)
START_PROLOG	START PROLOG	1 TASK PROC OR BLOCK
END_PROLOG	END PROLOGUE	1 TASK PROC OR BLOCK
START_EPILOG	START EPILOGUE	1 TASK PROC OR BLOCK
END_EPILOG	END EPILOGUE	1 TASK PROC OR BLOCK
PLIST	PARAMETER LIST	0 PARAMETER LIST DELIMITER
END_PLIST	END PARAMETER LIST	0 PARAMETER LIST DELIMITER
PARM	PARAMETER	2 PARAMETER SPECIFIER

The parameter list shall be specified at the call and in the prologue and epilogue. The parameters are:
At the call -
(ACTUAL-PARAMETER)
(FORMAL-PARAMETER)

IN PROLOGUE/EPILOGUE
(FORMAL-PARAMETER)
(PARAMETER-DUMMY)

Parameter lists shall be in declaration order.

START_AGG	START AGGREGATE	1 RECORD
END_AGG	END AGGREGATE	1 AGGROP
AGGREGATE OP		2 Value and component
RANGE_CHECK	CONSTRAINT CHECK	3 VALUE AND TWO CONSTRAINTS

This operator conditionally raises the constraint error exception and is used for sinple constraints. This operator is found common though it produces no result. Local optimization shall done by the code generator.

RAISE_CONSTR	RAISE CONSTRAINT ERROR NO OPERANDS	

This operator is used for the more complex constraint checks requiring formal IL with relationals and conditional branching.

ADD	ADD	2 NUMERIC
SUB	SUBTRACT	2 NUMERIC
MUL	MULTIPLY	2 NUMERIC
DIV	DIVIDE	2 NUMERIC
EXP	EXPONENTIATION	2 NUMERIC
SCALE	BINARY SHIFT	2 NUMERIC
ABS	ABSOLUTE VALUE	1 NUMERIC
MINUS	UNARY MINUS	1 NUMERIC
REM	REMAINDER	1 (INTEGER DIVIDE)
MOD	MODULUS	2 NUMERIC INTEGER

CONVERT	NUMERIC CONVERSION	1 NUMERIC
	The conversion is indicated by the type of the operand and the resulting type of the convert operator.	
AND	BOOLEAN AND	2 BOOLEAN
OR	BOOLEAN OR	2 BOOLEAN
XOR	BOOL EXCULSIVE OR	2 BOOLEAN
NOT	BOOLEAN COMPLEMENT	1 BOOLEAN
NEW	ALLOCATOR	1 TYPE OR LITERAL AGGREGATE
EQUAL	EQUALITY TEST	2 ANY TYPE
LESS	LESS TEST	2 SCALARS
GREATER	GREATER TEST	2 SCALARS
IN	MEMBERSHIP TEST	2 (VALUE AND CONSTRAINT SPECIFICATION)
	Negation indicator in operator.	
CONTROL TRANSFER		
CJUMP	CONDITIONAL JUMP	2 BOOLEAN AND LABEL
	TRUE/FALSE indicator in operator	
GOTO	UNCONDITIONAL GO	1 LABEL
CALL	PROC CALL	1 PROCEDURE
	Used for procedure for function reference	
EXIT	LOOP EXIT	2 LABEL AND LOOP IDENTIFIER
PRIM	PRIMITIVE OPERAND	1 SPECIFIES OPERAND
SUBSCRIPT	SUBSCRIPT	2 or MORE PRIM AND SUBSCRIPT

(This shall have a variable number of operands with the subscripts uncombined.

SLICE	ARRAY SLICE	3 PRIM SUBSCRIPTed PRIM or SLICE AND LOWER/UPPER BOUNDS

DEBUGGING:

ON	TURN FLAG ON	2 PHASE AND NUMBER
OFF	TURN FLAG OFF	2 PHASE AND NUMBER
DUMP	DUMP	3 PHASE, ADDRESS, NUMBER OF WORDS
LINE	LINE NUMBER	0
PRESET_REPL	REPLACE	2 USED FOR GENERIC DATA INITIALIZATION
REPL	REPLACE	2 SOURCE AND SINK
	SCALAR	
	BLOCK	
	AGGREGATE	
	ACCESS	
FUNC_RES	FUNCTION RESULT	1
TASK	TASK CREATION	1 TASK OPERAND
INFO	UNSTRUCTURED INFORMATION	

Type and number of words alwys can be ignored.
I/O calls shall be handled as normal calls on packaged procedures.

Prologues. TASKS PROCS AND BLOCKS

The 'PROLOG' and 'END_PROLOG' operators shall delimit the IL for the prologues. Every TASK, PROC and BLOCK shall have a prologue - the prologue may be empty. The code generated for a prologue shall be specified both explicitly in the IL and implicitly as a function of the symbol table information for the TASK, PROC or BLOCK.

The explicit IL for the prologue shall include the following:

Parameter list specification (PROCs only)
Data allocation/elaboration
Initial value assignment
Task initiation

The parameter list specification could be ignored for some targets. The default initial value assignments of input parameters shall be separate from the other initial value assignments. Target dependent local optimization should be done on the constant initial values.

EPILOGUES. TASKS PROCS BLOCKS

The 'EPILOG' and 'END_EPILOG' operators shall delimit the IL for the epilogue of each TASK, PROC and BLOCK. Every TASK, PROC and BLOCK shall have an epilogue - it may be empty. The code generated for the epilogue shall be controlled both explicitly in the IL and implicitly as a function of the symbol table information.

The explicit IL for the epilogue will include the following:

Parameter list specification (output parameter PROCs only)
Tasking control
Function result (PROCS only)
Heap space management
Stack space management
Return mechanism (PROCS)

Output parameters shall be handles in a target dependent way.

LOOP Statements

All LOOP statements shall be delimited in the IL with a BEGIN_LOOP and an END_LOOP IL operator; these operators shall have as a single operand the loop identifier.

An END_LOOP_LABEL operator shall follow the BEGIN_LOOP operator. The END_LOOP_LABEL operator shall have on label operand - the label shall be defined at the end of the loop.

An indefinite loop shall be expressed with a "GOTO loop_identifier" inserted before the END-LOOP.

The WHEN loop shall be expressed in the IL as an "IF ... THEN" statement with a "GOTO loop-identifier" inserted before the END_LOOP.

The FOR loop shall be expressed in the IL with the FOR and END_FOR IL operators or the FOR_LIST and END_FOR_LIST operators.

BEGIN_LOOP	1	LOOP IDENTIFIER
END_LOOP	1	LOOP IDENTIFIER -- loop delimiter
END_LOOP_LABEL	1	LABEL -- defined after the END_LOOP
FOR	4	(LOOP PARAMETER)
		(INITIAL VALUE)
		(INCREMENT)
		(FINAL VALUE)
END_FOR	0	-- FOR loop delimiter
FOR_LIST	2	(LOOP PARAMETER)
		(ENUMERATION TYPE) -- enumeration list
END_FOR_LIST	0	-- FOR_LIST loop delimiter

Subscripting

A subscript shall be expressed in the IL with a SUBSCR operator. The SUBSCR operator shall have a variable number of operands with a minimum of two operands; the first operand is the PRIMitive being subscripted and the second and following operands are the subscript expressions - one operand for each dimension.

Constraint checking may be interspersed within the SUBSCR operands.

Array Slices

Array slices shall be specified with the SLICE operator. The SLICE operator shall have three operands - the first operand shall be a PRIMitive specifying an array (this array may be subscripted and may itself be a slice); the second and third operands shall be the lower and upper founds of the slice.

TASK Creation/Elaboration.

Task creation and elaboration - that may result from a Task declaration, a task object declaration or a task access variable allocation - shall be expressed in the IL with the TASK operator. The TASK operator shall have a single TASK operand.

A description of the IL record structure is shown below.

Enumeration types for IL

```
type IL_OPERATOR is
     (START_PACKAGE, END_PACKAGE, START_TASK, END_TASK, START_PROC,
     END_PROC, START_BLOCK, END_BLOCK, START_PROLOG, END_PROLOG,
     START_EPILOG, END_EPILOG, START_PLIST, END_PLIST, PARM,
     RANGE_CHECK, RAISE_CONSTR, ADD, SUB, MUL, DIV, EXP, SCALE, ABS,
     MINUS, REM, MOD, CONVERT, AND, OR, XOR, NOT, EQUAL, LESS, GREATER,
     IN, NEW, CJUMP, GOTO, CALL, EXIT, PRIM, SUBSCRIPT, SLICE, LINE,
     REPL, PRESET_REPL, FUNC_RES, START_AGG, END_AGG, AGGROP, VALU, NEW,
     TASK, BEGIN_LOOP, END_LOOP, END_LOOP_LABEL, BEGIN_FOR, END_FOR,
     BEGIN_FOR_LIST, END_FOR_LIST, ON, OFF, DUMP, INFO, INTLIT, FLTLIT,
     CHARLIT, BOOLIT, STRLIT);
type IL_BASE_TYPE is
    (NULL,INTEGER,FIXED,FLOATING,BOOLEAN,CHARACTER,STRING,
    STRUCTURE);
type IL_SYM_PTR is access SYM;  -- symbol table entry pointer
```

IL Entry

```
type IL (ILOPR:IL_OPERATOR) is
    record
      case ILOPR is
         when
           ADD | SUB | MUL | DIV | EXP | SCALE | ABS | MINUS |
           REM | MOD | CONVERT | AND | OR | XOR | NOT | EQUAL |
           LESS | GREATER | IN | PRIM | SUBSCRIPT | SLICE =>
              IL_TYPE:  IL_BASE_TYPE; -- primitive type - INTEGER,
                        -- floating etc
```

```
    IL_SIZE:  INTEGER range 0 .. MAX_IL_SIZE;
        -- size of result
    IL_SCALE: INTEGER range -MAX_SCALE .. MAX_SCALE;
    IL_SIGNED:  BOOLEAN;
        -- true if signed ie. possibly negative
    IL_NEG:  BOOLEAN;   -- set true for negated relation
when VALU
  IL_VALU: INTEGER range 0 .. MAX_VALU_NO ;
        -- valu number

when PRIM =>
  IL_STD_OPND:  SUM_PTR;  -- symbol table entry

when LINE =>
  IL_LINE_NUMBER:  SRKEY;  record key line number

when INT_LIT =>
  IL_INT:  INTEGER;  -- converted INTEGER

when FLTLIT =>
  IL_FLOAT:  FLOATING;  -- converted real number

when CHARLIT =>
  IL_CHAR:  CHARACTER;  -- single character

when BOOLIT =>
  IL_BOOL:  BOOLEAN;

when STRLIT =>
  IL_STRP:  SYM_PTR;  -- symbol table character STRING
```

```
      when ON | OFF | DUMP =>
        IL_DBPH:  INTEGER;  -- phase number
        IL_DBNO: INTEGER;  -- debug options
        IL_DCOUNT: INTEGER;  -- debug count

      when others =>   -- other operator values of IL_OPERATOR

    end case;
  end record;
```

APPENDIX C - ADA RELOCATABLE OBJECT FORMAT

The primary result of an Ada compilation is a relocatable object. Several relocatable objects may be combined using the Linker to produce a composite relocatable object or a loadable object target-independent, relocatable object.

A relocatable object comprises several forms of information. This information and its purpose is listed below:

1. Identification - contains library unit/subunit name and version, compiler/linker version, compilation time stamp, and target identification.

2. Preamble - includes lists of defined names and addresses, external names, exception names, program sections with their sizes and attributes and a starting address if applicable. This data alone must be sufficient for the Linker to perform the allocation and resolution processing.

3. Unit-specifications - comprises all information produced by the compiler necessary to support library unit visibility, insure compilation order validation and perform library unit resolution. The names of other library units that have been compiled using this unit_spec is contained here.

4. Symbolic data - used by run-time facilities to support debugging, performance monitoring, data recording and subsequently data reduction. Included in the data are identifiers and their respective attributes, statement description information and flow graphs, compilation statistics and support for run-time statistics collection.

5. Concordance data describing library unit name references and definitions. This is used by the Linker at user option to produce a combined external symbol concordance.

6. Object data with relocation description information used by the Linker to produce an executable program image.

As part of the compilation and/or linking process, certain parts of the object module are copied, on option, into the active program library. Separate command utilities are provided which shall allow the placing, or replacing, of this data in a library (see Program Libraries).

The format of a relocatable object is defined below:

```
type PROC_TYPE is (ADA, LINKER, ASSEM);  -- creating processor
type LIB_TYPE is
          (SUBPRG, SUBPRG_BODY, PKG, PKG_BODY, SUBUNIT, COMPOSITE);
type NAME is STRING (1..72);
type VECTOR is array (INTEGER range <>) of BOOLEAN;
pragma pack VECTOR;
type COMPUTERS is (IBM370, INTERDATA_8_32);
type REL_ENUM is
          (IDENT, PREAMBLE, UNIT_SPECS, DEBUG, CONCORD, DATA, TERM);
type PROT_TYPE is (READ, READ_WRITE, WRITE, CODE);
type LOC_CTR is
     record
          LOC_NAME: NAME;
          SEG_NO: INTEGER;
          ADDR: INTEGER;
          SIZE: INTEGER;
          PROTECTION: PROT_TYPE;
          ATTR: BYTES;
     end record;
type DEFS is
     record
          DEF_NAME: NAME;
          DEF_ADDR: INTEGER;
     end record;
type PATH is
     record
          FROM,
          TO: PATH_INDEX;
     end record;
```


1243

```
type PATH_INDEX is range 1 .. MAX_PATHS;
type BLOCK_INDEX is range 1 .. MAX_BLOCKS;
type EXIT_CAUSE is (GO, IFF, LOP, CAS, PRC, EXCEPT);
type BLOCK is
     record
          BEG,
          EN: IL_INDEX;
          SUPER_EXIT: BOOLEAN;
          FOR_DOM,
          BACK_DO: PATH_INDEX;
          NO_FD,
          NO_FD: INTEGER;
          EXIT_DESCR: EXIT_CAUSE;
     end record;
type STMT is
     record
          PATH_TO,
          PATH_FROM: BOOLEAN;
          VARS_SET: INTEGER;
          OPT_INDICATOR: VECTOR (OPT_TYPE);
          SET_VARS: SET_VARS_INDEX;
          TIMING: DURATION;
          BEG_ADDR,
          IMPL_ADDR: ADDRESS;
     end record;
```

```
type REL (REL_TYPE: REAL_ENUM; UPPER_BOUND : INTEGER) is
     record
          NO_WORDS: INTEGER;
          CHECKSUM: VECTOR (1 .. TARGET_SIZE);
          case REL_TYPE is

          when IDENT =>
               LIB_UNIT: NAME;
               OBJ_NAME: NAME;
               PROCESSOR: PROC_TYPE;
               PROC_TIME, CREATE: TIME;
               TARGET: COMPUTERS;
               OBJ_VERS: INTEGER;

          when PREAMBLE =>
               NO_DEFS,
               NO_EXTS,
               NO_LOCS,
               NO_EXCS: INTEGER := UPPER_BOUND;
               DEF: array (1 .. NO_DEFS) of DEFS;
               EXCEPT_NAME: array (1 .. NO_EXCS) of NAME;
               LOCS: array (1 .. NO_LOCS) of LOC_CTR;
               EXT_NAME: array (1 .. NO_EXTS) of NAME;

          when CONCORD =>
               NO_ENTS: INTEGER := UPPER_BOUND;
               REF_NAME,
               LIB_NAME: array (1 .. NO_ENTS) of NAME;

          when UNIT_SPECS =>
               NO_TABS : INTEGER;
               TAB: array (1 .. NO_TABS) of SYM_TAB;
```

```
        when DEBUG =>
             NO_STMTS,
             NO_SYMBOLS,
             NO_SET_VARS,
             NO_BLOCKS,
             NO_PATHS: INTEGER := UPPER_BOUND;
             SYMBOLS: array (0 .. NO_SYMBOLS) of SYM_TAB;
             PATHS: array (0 .. NO_PATHS) of PATH;
             BLOCKS: array (0 .. NO_BLOCKS) of BLOCK;
             STMTS: array (0 .. NO_STMTS) of STMT;

        when DATA =>
             null;

        when TERM =>
             null;

    end record;
```

APPENDIX D - PROGRAM LIBRARIES

Program Libraries are used to provide a repository for interface information used as communication between the Compiler and Linker tools or successive invocations of these tools.

Additionally, information has been placed in the Program Libraries to significantly improve the performance of the Linker and to provide additional user oriented facilities. Both purposes shall be described in more detail further on.

The contents of a Program Library are presented below.

1. For each compilation unit, the Compiler shall distill information from the symbol table that represents the external interface specification of the compiled unit that is necessary to permit referencing by separate units of the visible entities of the compiled unit. Hereafter, this distillation is referred to as the "unit specs".

2. Derivation and location information shall also be maintained in the Program Library to identify the particular object version associated with a compilation unit, and the tool and input object versions (and associated unit-specs) used in its creation. Additionally, compilation and linking date and times shall be recorded to validate compilation order compliance.

3. Relocatable object preambles shall also be maintained in the Program Library. This information shall permit the Linker to complete its first pass (object requirement analysis, symbol resolution and allocation) by using solely the information from the Library. Only during the second pass when creating the executable object must the Linker access the required relocatable object. The primary benefit from this approach is the elimination of the opening of each object during the first pass, that has historically been an expensive operation in operating systems.

The Program Library to be used for compilation or linking shall be identified by default or by explicit naming in the compile or link command. At compilation when a "with" statement is encountered, the Compiler shall access the indicated unit-specs and add this information into the symbol table. Upon completion of a compilation, the unit-specs for the current compilation unit shall be entered into the Program Library. The unit-spec for a previous version of the compilation unit shall be superceded. A library utility shall allow the user to select a particular relocatable object version's unit-spec to be entered into the library.

When an executable object is to be created, the Linker is invoked. The relocatable objects to be used are accessed through the Library and the executable image is created. As part of this process, the Linker shall compare the date and time of each unit-spec with the date and time of all references to that unit-spec to insure valid compilation order. Any discrepancies shall be diagnosed by the Linker.

An additional feature of the Linker is to produce on user option a global concordance listing of references to unit-spec names. This listing shall allow a user to easily locate all compilation units that reference a visible entity of another compilation unit thereby simplifying large program maintenance procedures and eliminating unnecessary recompilations. To support this feature, the Compiler shall enter into the Program Library a list of names referenced by each compilation unit from a "with" ed compilation unit.

APPENDIX E - LISTINGS

The Compiler shall produce the following listings: source, dianostics, cross reference/attribute, object, statistics, environment, and system management. There shall be Compiler options to produce or suppress these listings. On the top of each page of each of these listings there shall be a header line containing: the page number, the date and time of compilation, the name of the compilation unit, and the version of the Compiler.

Source Listing

The Compiler shall produce a listing of the source statements it processes in the course of a compilation. There shall be various options to control the contents and format of the listing.

The listing itself shall be optional. The option to turn off the listing takes precedence over any list progmas in the source program. There shall also be an option to print the lines which are processed as the result of an INCLUDE pragma. This option shall also allow the printing of lines of included text to be suppressed, even if the source from the compilation unit is being printed.

The list pragmas shall only be honored in those portions of the source where the Compiler options specify that printing is to occur. Within such regions source shall be printed or printing shall be suppressed according to the list pragmas.

There shall be an option to produce a prettyprinted listing. This shall be independent of the option to produce a reformatted source file.

The Compiler shall assign statement numbers to each source statement it processes. There shall be an option to use these assigned numbers or the source keys in the listing.

Each page of the source listing shall contain column numbers.

Each line of the listing shall contain: the statement number, the source key number, the source line, the lexical nesting level, the compound statement nesting level, and an indication as to whether the line was included.

There shall be two options for diagnostics. One option shall control the level of messages to be printed. The other shall control the placement of the messages.

There shall be five levels of messages: note (N), warning (W), error (E), serious error (S) and fatal error (F), in order of increasing severity. On option the printing of notes or notes and warnings may be suppressed.

Error messages shall be printed on the first available line following the line on which the error was detected. If multiple errors are detected on one source line, each error shall be printed, one message to a line. For each error for which it is applicable, there shall be a pointer to the column or columns where the error occurred.

There shall be an option to print all diagnostics at the end of the listing, rather than interspersed with the source.

Error mesages, regardless of where they are placed shall contain the following: statement number or key, if applicable, an error severity indicator, an error message number, and the text of the message. The message text shall be a description of the error and may include inserts such as names or column numbers.

Assembly Listing

The assembly listing is produced by the assembler phase of the Compiler. It is more correctly called a pseudo-assembly listing sine the listing is produced by "disassembling" machine instructions obtained via the code file, rather than as the result of processing assembly language source.

The listing can be used for a number of purposes. Perhaps the most important use is an an aid in Compiler development and maintenance. The listing allows a visual inspection of the generated code. Such a listing can also be useful for debugging at the machine-level (such as on a target which does not have a source level debugger).

The listing itself is a side-by-side listing which contains both pseudo-assembler source and the octal (or hex) representation of the generated code. Since the listing is a human-readable rendering of machine code, it is, necessarily, largely machine dependent. Much of the processing, however, is the same for various targets. It is possible, therefore, to make the generation of the assembly listing largely table-driven in the interest of retargetability.

The format that shall be used shall be basically the same for all machines. The radix for the printing of numbers shall be octal or hexadecimal, depending on the natural radix for the target. The following types of lines shall appear in the listing: source, comments, error indicators, segment indicators, and machine code.

Comments

Comments shall be used to make the assembly listing more readable. They shall include indications that code belongs to prologs, exception handlers, etc.

Error Indicator

A line indicating that an error was present shall be included in the assembly listing in he code for the statement in which the error was detected.

Segment Indicators

Special lines shall delimit segments or control sections

Machine Code

Machine instructions shall be represented both symbolically and in the native radix. The following is a list of fields that shall be present in machine code line, together with an explanation of their contents.

Location - The offset from the start of the segment for this particular instruction. (Displayed in the native radix).

Contents - The contents of the storage unit(s) containing the instruction. (Displayed in the native radix. The contents field shall consist of subfields each of which contains an element of the instruction (e.g., op code).

Relocation - For each field in the contents that is relocatable, a flag shall be included in order to indicate the type of relocation. The following codes shall suffice for most targets:

blank	absolute
C	code
D	data
K	constants
X	external

Operand(s) - The address(es) of the operand(s) of the instruction shall be displayed, if the actual address differs from the address subfield in Contents. This can occur on a machine which uses base registers (e.g., the IBM 370).

Line The line number of the source statement that caused this code to be generated is displayed in decimal.

Label The user-defined or Compiler-generated label associated with this location is displayed. User-defined lables are represented by their source name. Compiler-generated labels are given generated names.

Op code - The mnemonic op code for this instruction, or the appropriate pseudo-op for data, is displayed.

Operands - The operand(s) appear in a pseudo-assembly language format. This shall be a standard form, based on the IEEE Microprocessor Assembly Language Standard. One known extension is allowing double indexing on the 370.

As was mentioned previously, the bulk of the formatting of the assembly listing shall be table driven. This requires tables that indicate where fields in the instructions are located and where they should be printed. The "assembled" portion of the listing shall be formatted from the instruction itself rather than directly from the code file, in order to minimize the chance of there being a discrepancy between the bit patterns printed out in the listing and the bit patterns that are stored in the relocatable object.

The "assembler source" portion of the listing shall be formatted fromthe code file, since there is information (such as symbol table pointers) that does not appear in the relocatable version of the instructions, but is necessary to produce symbolic names. This portion of the listing could be more target-dependent than the "assembled" portion, since there is more variety in assembler syntax than in instruction word formatting. In order to minimize the impact of retargetting, a standard format shall be used wherever possible, sacrificing compatability with local assemblers for ease of retargetability.

Attribute/Cross-Reference Listing

The Compiler shall generate combined attribute and cross-reference listings. Combining the two listings shall aid the user. In particular, it shall be easier to verify that unintended overloading resolution did not take place.

Several listing options shall be available.

It shall be possible to select an "attributes only" listing or an attribute/cross reference listing.

There shall be an option that will allow user control of the order in which entries are sorted. One choice shall be to have all of the object names sorted according to the collating sequence defined in the Ada Reference Manual. Another shall be to sort by scope and qualified name.

There shall be an option to allow the inclusion or exclusion of reserved wrds, attributes, operators from package standard, and constants.

There shall be an option to allow the inclusion or exclusion of unreferenced entities.

The following lists the attributes that shall appear in the attribute listing for each different type of entity. Those attributes that appear in upper case are defined in the Ada Reference Manual.

All entities

Name
Defining compilation unit

All Objects

Type
Representation (float, double float, etc.)
Location -- ADDRESS for static data; stack frame offset otherwise

F'54

Objects in records

FIRST_BIT

LAST_BIT

POSTION

Object of array type

FIRST

LAST

Enumeration Literals

Type

Representation

POS

PRED

SUCC

Labels

Location

Subprogram

Location

Parameter names

Parameter types

Result type (if function)

Inline

Linkage

Machine code

Code size

Data size

Packages

Location

Code size

Data size

Block

- Location
- Data size

Generic

- Formal parameters

Exception

Task or Task Type

- Location (task only)
- Body location
- Entries

All Types (except task type)

- BASE
- Limited
- Private
- Packed
- Controlled

Scalar types

- FIRST
- LAST

Fixed point types

- DELTA
- ACTUAL_DELTA
- BITS
- LARGE
- MACHINE_ROUNDS

256

Floating point types

DIGITS
MANTISSA
EMAX
SMALL
LARGE
EPSILON
MACHINE_RADIX
MACHINE_MANTISSA
MACHINE_EMAX
MACHINE_EMIN
MACHINE_ROUNDS
MACHINE_OVERFLOWS

Array type

FIRST
LAST

Set/Use

The set/use information consists of the source key for the declaration, and a sorted list of source keys for sets and uses. Sets and uses are sorted together in order of increasing source key. In the listing an indication shall be given as to which program unit contained the reference.

Statistics Listing

The Compiler shall produce a listing of various static statistics on option. Dynamic statistics are produced by the Debugger.

The level of statistics printed shall be controlled by an option. Brief statistics shall only include totals for the various groups. Verbose statistics shall include a breakdown by individual group members. These breakdowns shall provide both the number and precentage of occurrences.

The following statistics shall be printed:

- number of charcters
- number of lines
- average line length
- average number of leading blanks
- number of symbols (broken down by type)
- number of declarations (broken down by type)
- number of statements (broken down by type)
- number of compound statements (broken down by nesting level)
- number of procedures, functions, packages, tasks (by nesting level)
- calls (broken down by number of parameters)
- operands, operators in statement (broken down by number)
- operands, operators (broken down by type)
- operand locations (broken down by relative scope, offset)
- overloading (broken down by number of potential chocies)
- number of generic instantiations
- pragmas (broken down by type)
- errors (broken down by severity)
- errors (broken down by statement type)
- number of optimizations performed (broken down by type)
- machine operations generated (broken down by type)
- number of symbol table entries (broken down by type)
- space required for symbol table
- for each intermediate file:
  - size
  - number of entries (broken down by type)

System Management Listings

The Compiler shall produce various listings to facilitate its development and maintenance. Symbolic dumps shall be produced on option for the Compiler's internal files, including IL, code file, and the preset file. A symbol table dump shall be available, both for the entire table and for individual entries. A similar facility shall be provided for the Ada Debug Tables (ADTs).

Options shall be provided so that the major tables internal to the various Compiler phases can be dumped symbolically. In addition traces may provide a trace facility.

APPENDIX F - ADA RUN-TIME LIBRARY

Standard and Text Input-Output

Input-output facilities shall be fully defined in the packages INPUT OUTPUT and TEXT_IO.

The generic package INPUT_OUTPUT shall have to be instantiated by a user for each particular file ELEMENT_TYPE used; also, the TEXT_IO generic packages (INTEGER_IO, FLOAT_IO, FIXED_IO AND ENUMERATION_IO) for integer, floating, fixed and enumeration types shall have to be instantiated for each particular type used. The Compiler shall have no special or intrinsic knowledge of the predefined I-O packages.

Low level input-output shall be defined in the system packae LOW_LEVEL IO. The device types, data types and other I-O functions supported shall be target dependent and shall be supplied as part of the target system specification and requirements.

Tasking

Ada tasking functions shall be under the control of a run time tasking package. The compiler shall generate calls on the procedures and functions defined in the tasking package for all Ada tasking requirements. Reference Volume 2, KAPSE Framework, Section 3.3.6 Task Manager, for the definition of the tasking package.

Math and Data manipulation run time library routines

The math and general data manipulation routines required by the Compiler are listed below.

Exponentiation

Functions shall be required to perform integer to integer and floating in integer exponentiation.

```
function  Expon (I: INTEGER; J:NATURAL) return INTEGER;

function Expon (Fl: FLOAT; I:INTEGER) return FLOAT;
```

Whether explicit routines shall be needed for the various forms of integer and floating (short, regular, long) shall be target dependent.

Generalized Move routines

General move routines shall be defined for bit, character and word operands.

```
type BOOL_ARRAY is array (INTEGER range<>) of BOOLEAN;
type CHAR_ARRAY is array (INTEGER range<>) of CHARACTER;
type WORD_ARRAY is array (INTEGER range<>) of INTEGER;

procedure Move (Source: BOOL_ARRAY; Sink: out BOOL_ARRAY);
procedure Move (Source: CHAR_ARRAY; Sink: out CHAR_ARRAY);
procedure Move (Source: WORD_ARRAY; Sink: out WORD_ARRAY);
```

Generalized Compare Routines

General compare routines shall be defined for bit, character and word operands.

```
type RELATION is (EQUAL, LESS, GREATER);

function Compare (Rel   : RELATION;
                  Left  : BOOL_ARRAY;
                  Right: BOOL_ARRAY) return BOOLEAN;

function Compare (Rel   : RELATION;
                  Left  : CHAR_ARRAY;
                  Right: CHAR_ARRAY) return BOOLEAN;

function Compare (Rel   : RELATION;
                  Left  : WORD_ARRAY;
                  Right: WORD_ARRAY) return BOOLEAN;
```

Generalized Logical Operation Routines

General logical operation routines shall be defined for bit and word operands. Versions of these routines shall exist both as functions and as procedures.

```
type LOGICALOP is (AND, OR, XOR);

procedure Logic (Op  :     LOGICALOP;
                 La  :     BOOL_ARRAY;  -- first operand
                 Lb  :     BOOL_ARRAY;  -- second operand
                 Res : out BOOL_ARRAY); -- result operand

procedure Logic (Op  :     LOGICALOP;
                 La  :     WORD_ARRAY;  -- first operand
                 Lb  :     WORD_ARRAY;  -- second operand
                 Res : out WORD_ARRAY); -- result operand

function Logic (Op: LOGICALOP;
                La: BOOL_ARRAY; -- first operand
                Lb: BOOL_ARRAY) -- second operand
                return BOOL_ARRAY;

function Logic (Op: LOGICALOP;
                La: WORD_ARRAY; -- first operand
                Lb: WORD_ARRAY) -- second operand
                return WORD_ARRAY;
```

Generalized Concatenation Routines

General concatenation routines shall be defined for one dimensional bit, character and word arrays. Versions of these routines shall exist both as functions and as procedures.

```
procedure Catenate (La  :     BOOL_ARRAY;
                    Lb  :     BOOL_ARRAY;
                    Res : out BOOL_ARRAY);

function Catenate (La  :     BOOL_ARRAY;
                   Lb  :     BOOL_ARRAY) return BOOL_ARRAY;
```

```
procedure Catenate (La  :     CHAR_ARRAY;
                    Lb  :     CHAR_ARRAY;
                    Res : out CHAR_ARRAY);

function Catenate (La  : CHAR_ARRAY;
                   Lb  : CHAR_ARRAY) return CHAR_ARRAY;

procedure Catenate (La  :     WORD_ARRAY;
                    Lb  :     WORD_ARRAY;
                    Res : out WORD_ARRAY);

function Catenate (La  : WORD_ARRAY;
                   Lb  : WORD_ARRAY) return WORD_ARRAY;
```

Numeric Conversion Routines

Numeric conversion routines shall be required (eg. float to integer, integer to fixed, fixed to float etc.). Which conversions shall be performed with in-line code and which with library subroutine is target dependent and shall be separately specified for each target.

Time Utilities

Library functions to support the requirements of Package CALENDAR shall be needed.

```
function Clock () return TIME; -- current APSE time

function Time_Sum (a : TIME;
                   b : DURATION) return TIME;

function Time_Diff (a : TIME;
                    b : TIME) return DURATION;
```

Image and Value Functions

A number of conversion and formatting functions shall be defined to support the IMAGE and VALUE attributes. These routines shall also be used by the TEXT_IO library routines for the conversion and formatting requirements of GET and PUT.

Parameter Array Addressing Routines

A formal parameter array with unconstrained array bounds shall be passed as an array packet. The packet shall contain the address of the array and for each dimension the lower bound, upper bound and the dimension multiplier.

A number of array addressing routines (one, two, three and n-dimensional) shall be defined which shall perform boundary checking and array address calcuation given the actual subscripts and the array packet.

Where bounds checking is suppressed, array address calculation shall be generated as in-line code even for unconstrained parameter arrays.

```
type ARRAY_BOUNDS is
     record
          Lower_Bound: INTEGER;
          Upper_Bound: INTEGER;
          Multiplier: INTEGER;
     end record;

type ARRAY_PACKET_1 is
     record
          Location: INTEGER; -- base address
          Bounds: ARRAY_BOUNDS;
     end record;
```

```
     type ARRAY_PACKET_2 is
          record
               Location: INTEGER; -- base address
               Bounds_1: ARRAY_BOUNDS;
               Bounds_2: ARRAY_BOUNDS;
          end record;

     type ARRAY_PACKET_3 is
          record
               Location: INTEGER_1;-- base address
               Bounds_1: ARRAY_BOUNDS;
               Bounds_2: ARRAY_BOUNDS;
               Bounds_3: ARRAY_BOUNDS;
          end record;

--A(I)

     function Array_Address (Apl : ARRAY_PACKET_1;
                             I   : INTEGER) return INTEGER; -- address

-- A(I,J)
     function Array_Address (Apl : ARRAY_PACKET_2
                             I,J : INTEGER) return INTEGER;  --address

-- A(I, J, K)
     function Array_Address (Apl   : ARRAY_PACKET_3;
                             I,J,K : INTEGER) return INTEGER;  --address
```

Volume 6

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

MAPSE Linker

Prepared for

Rome Air Development Center
Griffiss Air Force Base, NY 13441

Contract No. F30602-80-C-0292

267

TABLE OF CONTENTS

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This specification establishes the design, development, and test requirements for the MAPSE Tool Set member, the Linker. The purpose of this specification is to define the Linker being designed as part of the Ada Integrated Environment contract for RADC. This document will serve to communicate the functional design decisions that have been adopted and to provide a baseline for the detailed design and implementation phase and to identify the interfaces between the Linker, the KAPSE system and the user.

1.2 FUNCTIONAL SUMMARY

The Linker is the MAPSE tool which is used to link several relocatable objects into a single object for loading purposes or for further linking. The Linker will permit the structuring of programs into multi-level overlays, it will support multiple location counters, resolution of external references, specification of symbolic equates and name definition, allocation of stack and heap space, relocation of address references, and program stub generation.

270

SECTION 2 - APPLICABLE DOCUMENTS

The following documents form a part of this specification to the extent specified herein.

2.1 PROGRAM DEFINITION DOCUMENTS

1. Reference Manual for the Ada Programming Language, July 1980
2. Requirements for Ada Programming Support Environment, "STONEMAN," February, 1980.
3. Statement of Work, Contract No. F30602-80-C-0292, 80 Mar 26.

2.2 INTER-SUBSYSTEM SPECIFICATIONS

4. System Specification for the Ada Integrated Environment.
5. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.
6. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.
7. Volume 3, Computer Program Development Specification for CPCI APSE Command Language Interpreter.
8. Volume 4, Computer Program Development Specification for CPCI MAPSE Configuration Management System.
9. Volume 5, Computer Program Development Specification for CPCI MAPSE Compiler.
10. Volume 7, Computer Program Development Specification for CPCI MAPSE Editor.
11. Volume 8, Computer Program Development Specification for CPCI MAPSE Debugger.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

12. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.
13. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

14. Ada Compiler Validation Implementers' Guide, October 1, 1980.

272

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section provides the general description, identifies the external and internal interfaces, provides the functional requirements and presents the internal characteristics of the Configuration Item identified as the Linker.

3.1.1 General Description

The Linker is used to combine one or more relocatable objects, including any referenced objects from the user program library (or from higher level project or system libraries), and produce a single relocatable object or load object and associated informational listings.

3.1.2 Peripheral Equipment Identification

The Linker will interface with peripheral equipment only as a potential source for its directives. However, since this interface will be through the standard Ada I/O package, the devices will be transparent to the Linker.

Although the Linker is largely machine independent, the Linker will have a dependency on the target calling convention and location counter characteristics. Initially the computer equipment of concern is identified as the IBM VM/370 and the Interdata 8/32.

3.1.3 Interface Identification

The Linker interfaces are identified as the APSE command language interpreter (ACLI), the KAPSE Data Base System (KDBS), the Ada Compiler, the Loader and the user through the Linker Directive Language.

3.1.4 Function Identification

The Linker is functionally divided into two main phases in the linking process. These two phases are identified below.

3.1.4.1 Program Structure Analysis and Program Order Validation

The first phase of the Linker validates the order of compilation requirements of the objects being linked and produces a fully defined program structure of the output object. The program structure includes

segment allocation, symbol resolution and any required library unit inclusion. The first phase is performed through access to the program library only and does not require access to the relocatable objects being linked.

3.1.4.2 Object Creation and Listings

The second phase of the Linker accesses the relocatable objects being linked, relocates and forms the memory image of the linked segments and writes the output object. The program library is updated at this time and the map and concordance listings are produced.

3.2 FUNCTIONAL DESCRIPTION

This section describes the functions of the Linker, the program and equipment relationships and interfaces identified above, and the input/output utilization in the Linker.

3.2.1 Equipment Description

The computers upon which the Linker will be initially hosted are the IBM VM/370 and the Interdata 8/32.

3.2.2 Computer Input/Output Utilization

Linker directives are assumed to exist in a standard text object or in the std_in file. Listing output is written to the std_out file and diagnostic messages are directed to the std_err file.

3.2.3 Computer Interface Block Diagram

The relationship between the Linker and the MAPSE system is shown in Figure 3-1:

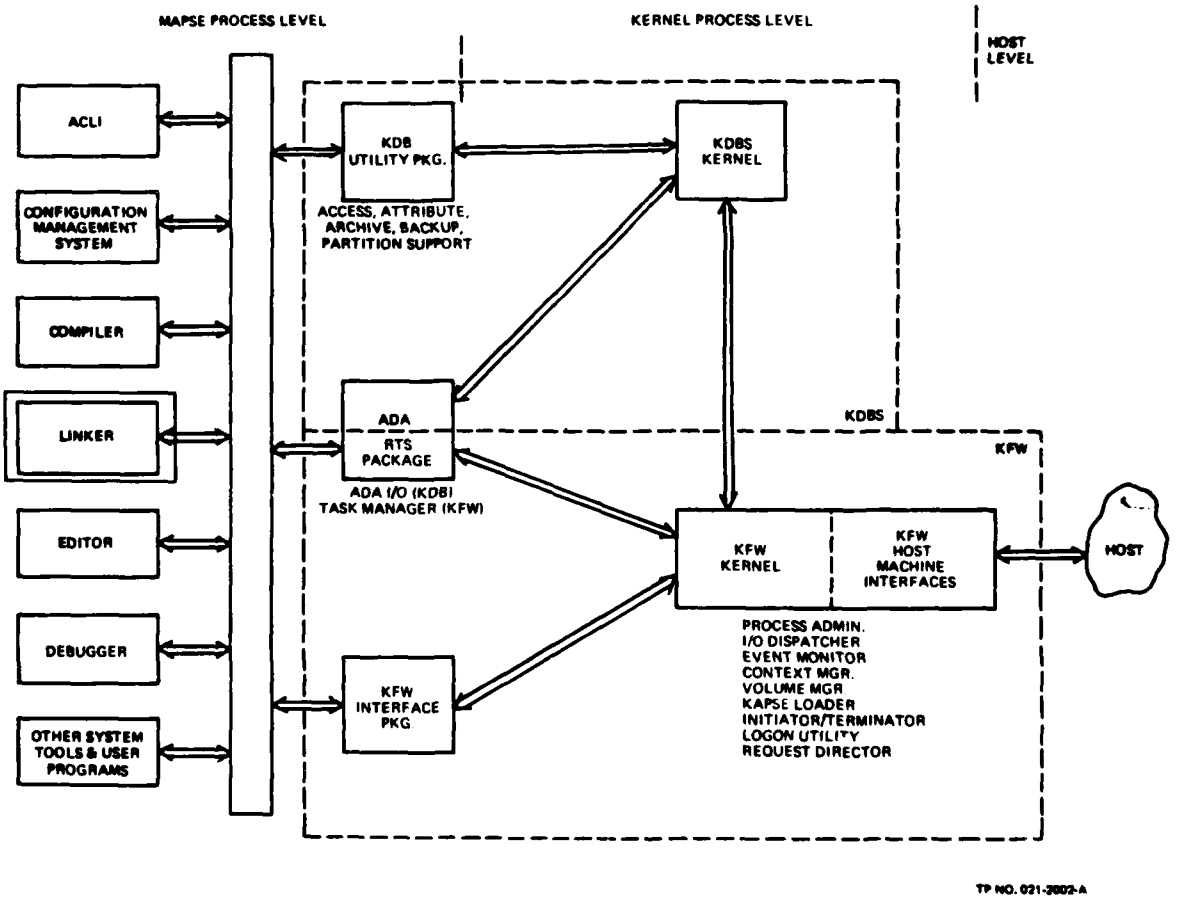


Figure 3-1. Interface Diagram

275

3.2.4 Program Interfaces

This paragraph identifies the Linker interfaces and their purposes.

3.2.4.1 APSE Command Language Interpreter (ACLI)

The Linker is invoked through the ACLI either from a user command at a terminal or from another MAPSE tool.

3.2.4.2 KAPSE Data Base System (KDBS)

The KDBS is called through the Standard I/O package to perform input/output on program libraries and relocatable objects. The KDBS is also called through the Standard I/O package to perform directive input and to output various Linker listings to std_out and std_err.

3.2.4.3 Linker Invocation Interface

The Linker will be invoked through the ACLI. The Linker command in general contains the following information:

1. the objects to be linked together and their structure

2. the name to be given to the linked object

3. linker options

The linker command exists in two forms, the first form is where the objects to be linked together are explicitly listed, the second form is where the objects to be linked and the desired program structure are specified with linker directives. With the first form the specified objects are linked together in a single unstructured segment.

The ACLI linker command is shown below.

1. LINK ((obj-1 {,obj-2 ...}), NAME=>obj-name, OPT=>options, ENTRY=>ext-name);

2. LINK (link-dir, NAME=>obj-name, OPT=>options)

The parameter names are optional.

2-76

The first command specifies a set of relocatable objects to be linked together into a single segment, the second command specifies the name of a standard text object containing linker directives which specify the modules to be linked together and the desired program structure. The link command directives are described in Appendix A.

Parameter Definitions

obj-n: APSE relocatable object name.

link-dir: APSE text object name containing Linker directives.

NAME: The name to be given to the output object module may be specified. The default name is obj-1'REL or obj-1'XQT for a partial link or a full link respectively.

ENTRY: The external symbol which is to be the entry point of a load module may be specified. The default is the first compilation unit name in the link.

OPT: The Link options are expressed as a series of single letters which may appear in any order.

The linker options are shown below.

Option	Code	Meaning	Default
Object Type	P	Partial Link	Load Object
Listing	L	Map Listing	No Map Listing
Concordance	C	Concordance	No Concordance Listing
Debug Tables	X	Exclude Debug Tables	Debug Tables Included
Visibility	V	Full Visibility for external symbols	Only unreferenced external symbols made visible

Thus with the full default options an executable load object would be produced with the Debug tables included, the only visible external symbol would be the entry point and neither a map nor a concordance listing would be produced.

277

3.2.4.4 Linker Directive Language

Linker directives may be used to define the structure and allocation of a linked object. The Linker directive language is defined in Appendix A.

3.2.5 Function Description

The Linker is invoked through the ACLI, either directly from a terminal or from another tool, with a 'LINK' command with parameters. The parameters specify the Linker options, the name of the output object to be written and the actual objects to be linked. The objects to be linked can either be explicitly named in the LINK command (for a simple single segment program structure) or can be specified through Linker directives. Linker directives specify the content, allocation and structure of the program to be linked. The output of the Linker can either be a relocatable object - which can be included in another Link, or a Load object - which can be loaded and executed. The Linker performs the final checking for the order of compilation requirements and optionally produces a map and concordance cross reference listing of the linked object.

The following paragraphs outline the functional Linker requirements which are needed to provide a useful, user-oriented development system for embedded computer applications.

3.2.5.1 Basic Linking Functions

The Linker will permit the structuring of programs into multi-level overlays, it will support the use of multiple location counters, resolution of external references, specification of symbolic equates and name definition, allocation of stack and heap space, relocation of address references, and program stub generation.

3.2.5.2 Partial Linking

To facilitate the development of large systems, the Linker will permit collection of several programs into a larger but still relocatable program. The format of the output program will be the same as the input program produced by the Ada compiler. A user may select those entry points that are to be visible in the output program and the names these points are to be

known by. External references satisfied by entry points contained in the programs being combined will be resolved; unresolved external references will be passed on in the program produced.

3.2.5.3 Debug Table Processing

Although the symbolic debug tables produced by the compiler for use by the Debugger will not be treated as resident location counter data, any addresses occurring in these tables will be resolved and relocated like the references in the object program code and data.

3.2.5.4 Object Version Genealogy

The Linker will support the MAPSE system in maintaining the history attributes of the object program produced. In addition to the information found in the input programs, the Linker will include the list of the programs being linked to create the new program and the version of the Linker being used.

3.2.5.5 Compilation Order and Version Validation

The Linker will be the tool with the ultimate responsibility to inform the user of compilation order violations and possible interface conflicts.

3.2.5.6 Linker Listings

The Linker will produce user-oriented listings describing the allocation of the various program location counters and entry points. Included in this listing will be any attributes of the various location counters such as read-only, read-write, shareable, self-relocating, instructions etc. The user may also select a cross-reference listing of the external references.

3.2.5.7 Symbol Definition

In the development of retargetable and rehostable software, such as the MAPSE itself, there is often a requirement to reference machine dependent subprograms having different names but with otherwise identical interfaces and functions. In the Debugger for example, the mechanism for implants will be target dependent and will also depend upon whether the program being debugged is a host program or a simulated target program.

Within the Debugger, calls will be made to the "IMPLANTER" subprogram. Several implant subprograms will exist with identical interfaces having names corresponding to the target. The 'EQUATE' directive is provided to allow two symbols to be equated at link time.

For example, when linking a Debugger for a particular embedded computer, the specific implanter (e.g., embedded_IMPLANTER) could be included and equated to "IMPLANTER" with the EQUATE directive. Absolute equates shall also be allowed to support the referencing of hardware dependent absolute locations.

3.2.5.8 Boundary Alignment

The Linker shall allow the specification of boundary alignments for any externally relocable element of the link such as an object, or a loction counter. The boundary alignment will be expressed as an absolute address or as some function of the "next available" location, such as double-word alignment, next byte, or next page.

3.2.5.9 Object Placement and Stub Generation

The Linker shall allow the user to completely specify the structure and order of programs in a linked program but will not require the user to do so. The Linker will automatically include any required objects in an appropriate position in the linked program. For any included object which is identified as a "stub" the linker shall supply a dummy procedure which will execute a return.

3.3 DETAILED FUNCTIONAL REQUIREMENTS

The linking process is divided into two major functional areas. The first validates the compilation order of the program being linked and builds a complete program structure with all addresses computed and external references resolved. The second function creates and writes linked objects for loading or further linking (partial link) and updates the program library. These functions will be described in the following sections.

780

3.3.1 Program Structure Analysis - Introduction

The first function performed in linking is validating the compilation order of the objects being linked and computing the allocation structure of the linked objects. This involves accessing the library unit-specifications of all objects included in the link. These specifications are available in the user program library or in some other higher level library visible to the user - a project or system library for example.

3.3.1.1 Inputs

The inputs to the Linker are the linker command, the optional linker directives, the program libraries available to the user and the relocatable objects specified in the link.

3.3.1.2 Processing

Initially, the Linker processes the input directives and creates a program structure description table which defines the program structure as explicitly directed by the user. (In the absence of directives the program structure is a single segment.) For each segment the program structure table contains a list of all the objects in that segment and a linked list of all the segments or overlays immediately contained in that segment.

3.3.1.2.1 Program Library Analysis

For each object, the relocatable object preamble record in the program library is accessed to obtain a list of the externally defined names and their relative addresses and the sizes of the location counters within the object. This information is attached to the program structure table. The derivation information in the program library is accessed to obtain the date and time of each module's compilation. An object which is identified as a "stub" will be allocated code space for a dummy return procedure.

The unit specification of each module in a segment is accessed from the program library to obtain a list of each library unit specified in a "WITH unit-name" of the module's context specification - these are called the "implied" library units and their processing is described below.

First, the date and time of compilation of the implied library units is checked to validate the required order of compilation - diagnostics will be issued where the required order is violated.
Secondly, for the Load object option (full link as opposed to a partial link) the implied library units are actually included in their appropriate segments in the overlay structure as described below.

Note: With the partial link option, the implied library units are not included and references to these units will remain unresolved in the output object. These implied library units will be included when the partially linked object is linked into a fully linked Load module. In this way common library units will be properly shared in the final program structure when referenced from several independent partial links.

3.3.1.2.2 Implied Library Unit Inclusion

Each implied library unit (specified in the context "WITH unit-name" and also referenced in the object) is added to the list of objects needed in the current segment unless it is already there or is visible in a higher level visible segment.

Each implied library unit is also added to the elaboration list of the segment unless it is already there or is visible in the elaboration list of a higher level segment. The elaboration list built during the first phase of linking is an unordered list, associated with each segment, of every library unit in that segment requiring elaboration at execution time.

As each library unit-name is added to the elaboration list for a segment, its unit specification is accessed from the program library and all of its implied library units are associated with the elaboration list entry (this will be used to establish an order for library unit elaboration). In addition, the normal library unit processing described above is performed for each of the newly implied library units. In this way all of the library units required by the program being linked are included.

3.3.1.2.3 Library Unit Promotion

Before a library unit is added to the list of objects required in a segment, the program structure is analyzed and if that library unit is implicitly required (as opposed to being explicitly included) in some other segment of the program, then the object is promoted to the lowest level segment common to both segments requiring the object. This process will necessitate the promotion, in a similar way, of any object implicitly required by the promoted object. When an object is promoted to a higher level segment, the elaboration lists of the segments are updated.

When an object is explicitly included in a segment with the INCLUDE directive, the object will not take part in the promotion process.

3.3.1.2.4 Allocation and Resolution

Address space allocation is performed on each location counter across the entire program structure and all externally defined symbols within each included object are given their final relative address value.

3.3.1.3 Outputs

The final output of the first phase of the linking process is the program structure table.

The program structure table will be used by the second phase of the linking process It will contain complete definition and allocation information for the segment/overlay structure of the linked program.

The entry for each segment in the structure table contains the segment's name and number, the allocation of each of its location counters, a pointer to a chain of each of its immediately subordinate segments, a pointer to a chain of its externally defined symbols and a pointer to its parent segment - these linked lists define the program allocation and overlay structure.

283

3.3.2 Linked Object Creation

The second major function performed by the linker is actually producing the linked object, updating the program library and printing the Map and Concordance listings on option.

3.3.2.1 Inputs

The inputs to the second phase of the linking process process are the program structure table (produced by the first phase), which fully describes the structure and content of the linked program being built, and the actual relocatable ojects making up the program. It is during this phase that the relocatable objects are accessed, relocated and combined as the linked object.

3.3.2.2 Processing

This phase of the linker produces and outputs the linked object.

This output object is OPENed for output; if the object does not exist it is CREATEd.

The identification record and unit specification record are produced and written to the output object. The unit specification record will describe the main program entry point specification only. Both of these records will be used to update the program library.

The preamble record is built and output. The preamble record will include a program structure table. The program structure table describes the segment/overlay structure of the linked object and separately identifies and describes each segment in the program.

For a partially linked program, the program structure table is used when the module is included in larger linked program. For a load object, the program structure table is used to build the Segment-Load-Table that is used at execution time for the loading and elaboration of the program segments. The preamble record will also be used to update the program library.

3.3.2.3.1 Segment Processing

Each segment in the linked object, which may contain one or more relocatable objects, is processed separately in the order of the overlay structure. Each relocatable object module in the segment is read, the object data records are accessed (this includes the symbolic debugging table records unless they were excluded) and an image of the included object is built with all possible relocation being applied. If the object module is identified as a "stub" the Linker shall supply a dummy procedure for that module which shall execute a return.

Note. The processed object may still contain unresolved external references (resulting from a partial link for example) and still contains general location counter relocation information to support partial linking (where each location counter may require further relocation when the partial link is included in another program) and also to support the possible requirement on some targets for dynamic relocation at load time.

As each segment is processed, its relocated object text is written to the object module being created. If the concordance listing option were specified, the concordance records of each object module are accessed and written to a temporary concordance file. When the link is complete, the concordance file will be read and sorted and a concordance listing printed.

If the map listing option were specified, a program map will be printed using the information contained in the program structure table.

3.3.2.2.2 Load Object Initiator Routine

When the Load option is specified, a system supplied root segment is included in the program to support initiation, segment loading, elaboration and execution of the program. This root segment will include an "Initiator" routine and a Segment Load Table. The "Initiator" routine and the Segment Load Table are described in Appendices B and C.

3.3.2.2.3 Elaboration Routine

When the Load option is specified the Linker will create an "elaboration procedure" using the elaboration list associated with each segment. The elaboration procedure will contain a standard procedure entry, a procedure

285

call to every elaboration prologue routine in the segment and a standard procedure exit. This elaboration procedure (which will be pure code) will be inserted into the code location counter of the linked segment and its address will be entered in the segment load table (see Appendix C). When segments are loaded this elaboration procedure may be executed as a parameterless procedure and it will perform the required elaboration of all the library units in the segment in the proper order.

3.3.2.3 Outputs

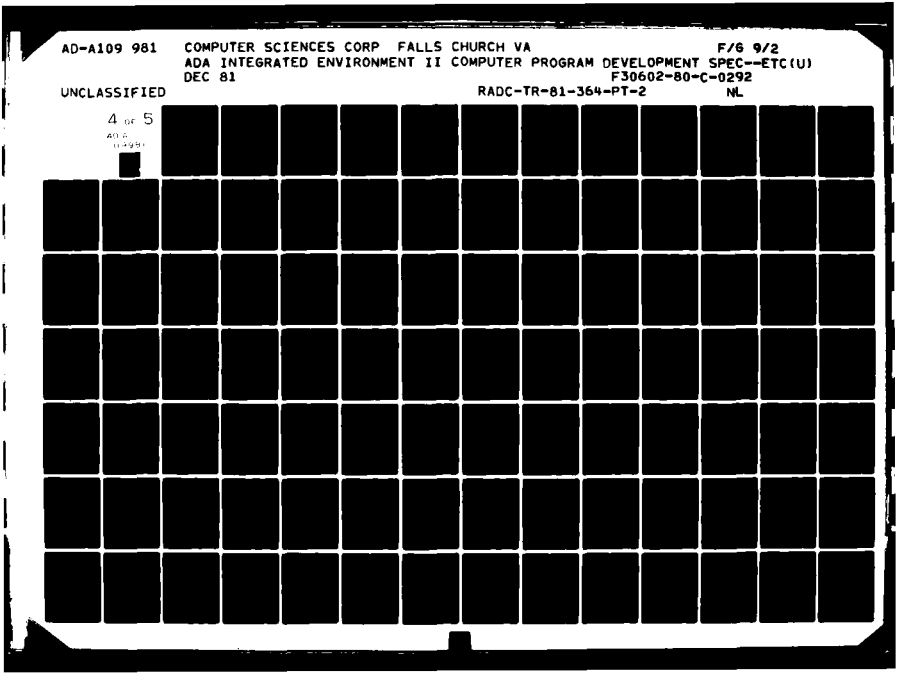
The linker produces the following outputs.

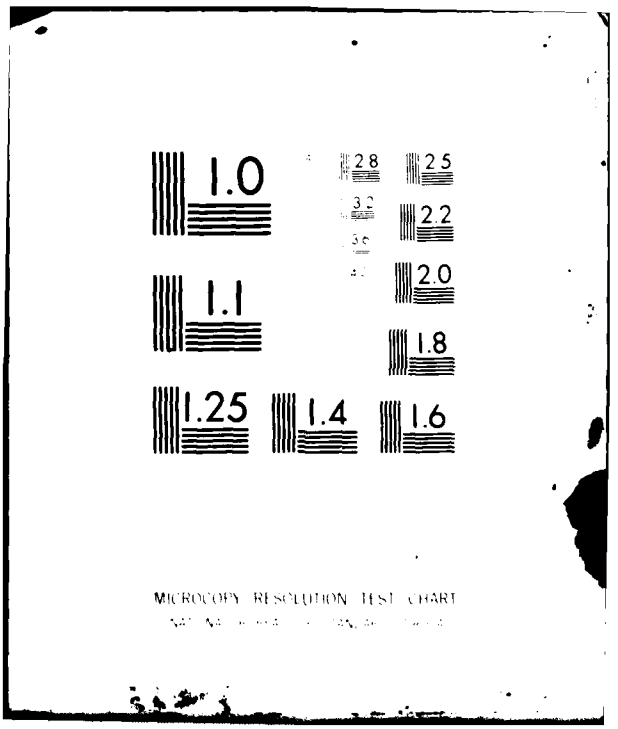
A linked relocatable or Load object.

Updates to the program library reflecting the newly linked object.

An optional printed map of the linked program structure.

An optional concordance listing of the linked object showing the library-unit cross reference requirement of ther full program.


AD-A109 981
COMPUTER SCIENCES CORP FALLS CHURCH VA
F/G 9/2
ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPEC--ETC(U)
DEC 81
F30602-80-C-0292
UNCLASSIFIED
RADC-TR-81-364-PT-2
NL
4 OF 5


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

3.4 ADAPTATION

This section describes the requirements of the Linker with respect to system environment, system parameters, and system capacities.

3.4.1 General Environment

Not applicable.

3.4.2 System Parameters

Not applicable.

3.4.3 System Capacities

The size of the memory partition allocated to the Linker will affect its performance - the Linker will be organized to page unit data into limited space, and to take advantage of dynamic memory allocation, when available.

3.5 CAPACITY

Not applicable.

SECTION 4 - QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the Linker. The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the Linker. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data are processed. For example, a review of hard copy test data might be used to verify that the values of specific parameters are correctly computed.

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

Table 4-1. Test Requirements Matrix

SECTION	TITLE	INSP.	ANAL.	DEMO.	DATA.	SECTION NO.
3.3.1	Program Structure Anal.		X		X	4.2.1,4.2.3
3.3.2	Linked Object Creation	X			X	4.2.1,4.2.3

4.1.1 Subprogram Testing

Following unit testing, individual modules of the Linker shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2. Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

289

CPCI testing shall be performed on all development software of the Linker. This specification presents the performance criteria which the developed CPCI must satisfy. The correct performance of the Linker will be verified by testing its major functions. Successful completion of the program testing that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3 System Integration Testing

System integration testing involves verification of the integration of the Linker with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the Linker performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

The Linker is a basic tool of the MAPSE system. After being unit tested, its major testing will be performed through its actual use in the development of the MAPSE system. Formal testing of the Linker is described below.

4.2.1 Inspection

Linked objects shall be validated by utilizing dump facilities and inspecting the output.

4.2.2 Review of Test Data

Each Linker command shall be tested by preparing scripts and comparing the output of the Linker with the expected test output. Specific programs and link directives will be developed to individually test every Linker function.

4.2.3 Special Tests

The MAPSE system itself, the Ada compiler and the other tools will be considered an extensive test case for the Linker. An operational system and compiler will verify the linking function here. The Ada compiler validation test cases used for compiler acceptance will also verify the linking functions for these programs. Special programs will be developed to specifically test the Linker's handling of error situations. This will include not only invalid Link directives and program structures but also order of compilation violations. Programs will be developed to verify any capacity limitations imposed by the Linker and to measure Linker performance.

4.3 ACCEPTANCE TEST REQUIREMENTS

Acceptance testing shall involve comprehensive testing at the CPCI level and at the system level. The CPCI acceptance tests shall be defined to verify that the Linker satisfies its performance and design requirements as specified in this specification. System acceptance testing shall test that the MAPSE satisfies its functional requirements as stated in the System Specification. The Linker is such a basic tool of the MAPSE system that delivery of a MAPSE with its tools will effectively constitute delivery of the Linker.

These tests shall be conducted by the CSC/SEA team and formally witnessed by the government. Satisfactory performance of both CPCI and system acceptance tests shall result in the final delivery and acceptance of the MAPSE system.

Results of all tests will be made available and the results of the capacity and performance testing will be incorprated in the Linker User Manual.

SECTION 5 - DOCUMENTATION

5.1 GENERAL

The documents that will be produced during the implementation phase in association with the Linker development are:

1. Computer Program Development Specification
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. Users Manual
6. Retargetability/Rehostability Manual
7. MAPSE Tools Reference Handbook

5.1.1 Computer Program Development Specification

The final MAPSE Linker B5 Specification will be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II. A single document will be prepared that defines the linker's functional capabilities and interfaces. Any dependencies on the host and target will be addressed in the document. Additionally, characteristics of potential hosts and targets which have had impact on the B5 specificiation will be presented.

5.1.2 Computer Program Product Specification

A type C5 specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document will be used to specify the linker design and development approach for implementing the B5 specification. This document will provide the detailed description which will be used as the baseline for any Engineering Change Proposals. A single C5 will be produced for the linker with different sections addressing the dependencies of the two host computers.

5.1.3 Computer Program Listings

Listings will be delivered which are the result of the final compilation of the linker. Each compilation unit listing will contain the corresponding source, cross-reference and compilation summary. The source listing will contain the source lines from any INCLUDEd source objects.

5.1.4 Maintenance Manual

A Linker Maintenance Manual will be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the Linker to be maintained by other than the developer. The documentation will be structured to relate quickly to program source. The procedures required for debugging and correcting the Linker will be described and illustrated. Sample run streams for compiling Linker components, for relinking the Linker in parts or as a whole, and for installing new releases will be supplied. The data base will be fully documented with pictures of record layouts where appropriate and data algorithms explained.

The Maintenance Manual will be organized with a standard outline and separate parallel sections will be delivered which address the tailoring of the Linker to a particular target or host computer. Debugging aids which have been incorporated as an integral part of the Linker will be described and their use fully illustrated. Special attention will be given to the description of the maintenance mode operation of the Linker used to aid in the pinpointing of Linker problems.

5.1.5 Users Manual

A Users Manual shall be prepared in accordance with DI-M-30421 which will contain all information necessary for the operation of the Linker. Because of the virtual user interface presented by the CLI and the Linker, a single manual is sufficient for all host computers. Sample linker listings will be included in the manual.

A complete list of all Linker diagnostic messages will be included with supplemental information chosen to assist the programmer in locating and correcting Linker directive errors.

5.1.6 Retargetability/Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual will be prepared which describes the procedures for retargeting the Linker to a different computer and integrating this new Linker into the system.

5.1.7 MAPSE Tools Reference Handbook

A MAPSE Tools Reference Handbook will be produced as part of Ada Integrated Environment contract. The Linker section of the handbook will contain syntax definitions and examples of the LINK command and the Linker directives.

APPENDIX A - DIRECTIVES

Linker Directives

The Linker function can be controlled with directives that specify the relocatable objects to be included and the structure of the linked program to be built. These directives may be input from std-in or from a text object.

The linker directives allow the specification of a multi-level overlay segment structure, support multiple location counters and allow heap and stack space allocation.

The individual linker directives are defined below, this is followed by an example of a linker directive object which builds a simple overlay program structure.

1. Relocatable object Inclusion

   INCLUDE obj-1, {,obj-2 ...}

   The 'INCLUDE' directive is used to specify the inclusion of one or more relocatable objects into the current segment. The relocatable objects may have been produced by the Ada compiler or may have been produced by the Linker as a partial link. Any number of 'INCLUDE' directives may be used to specify the objects to be included in a segment.

   The syntax of 'obj-n' is the KDBS object name optionally followed by a parenthesized list of location counter names or numbers. Within the list the location counters will be separated by commas.

2. Segment Identification Overlay Control

seg-name: SEGMENT [previous-seg-name]

The 'SEGMENT' directive is used to establish and identify a segment into which relocatable objects are to be INCLUDED. If the directive has the name of a previously defined segment as an operand, the origin of the newly established segment will be set to the location of the operand segment, thus creating an overlay on that previous segment.

3. Name Equates

name: EQUATE value

The 'EQUATE' directive is used to assign the specified value to a name. The value may be a constant integer expression or may be relative (+ or -) to some previously defined symbol.

For Example:

PRINTER: EQUATE 65776

ALPHA: EQUATE ALPHA_EXPERIMENTAL

When a symbol is explicitly defined with the 'EQUATE' directive, it will override the symbol definition in a library unit.

4. Data Reserve

[name:] RESERVE value

The 'RESERVE' directive is used to reserve a specified number of data storage units at the current segment location. The 'value' operand must be an integer constant expression or have been previously defined as an integer. If a name is used, it is made an external symbol.

5. Heap Space Specification

[name:] HEAP value

The 'HEAP' directive is used to both allocate the heap space at a particular location and to specify the size of the heap space. If a name is specified, the heap space will be allocated at the current location in the current segment. If no name is specified, the heap space will be allocated by the Linker. If a value of zero is specified, the size and location of heap space will be established at program invocation in a target dependent manner. If no 'HEAP' directive is used, the default allocation and size of heap space will be used - this will be target dependent. 'Value' must be an integer expression.

6. Stack Space Specification

[name:] STACK value

The definition of the 'STACK' directive is analagous to the 'HEAP' directive. A 'STACK' directive will cause the stack and heap space to be separately controlled. In the absence of an explicit 'STACK' directive, the stack and heap space will be shared.

7. Entry Name

[exname:] ENTRY name

The 'ENTRY' directive is used to identify 'name' as the entry point, of a main program, this is the location at which the program will start when initiated, 'name' must be externally defined. If 'exname' is specified, exname is used as the externally defined name in the output object.

8. Alignment

ALIGN type

The 'ALIGN' directive is used to perform allocation alignment on the immediately following segment or object, 'type' will be separately specified for each target.

9. Origin

ORIGIN value

The 'ORIGIN' directive is used to control segment allocation. The value must be a integer constant expression or relative (+ or -) to some previously defined symbol. The origin of the segment or object which follows will be established at the specified value.

10. Establish a Limiting Address

name: LIMIT seg-1 {,seg-2 ...}

The 'LIMIT' directive is used to identify the end of a set of segments and associates the specified name with the largest of the location counter values of the set of segments specified as

operands. The name can be used as the operand of a 'SEGMENT' directive to force a segment to be established at the end of the largest of a set of overlay segments. Each seg-n operand must be the name of a previously defined segment.

11. Library Update Name

LIBRARY lib-name

The 'LIBRARY' directive is used to specify which program library is to be updated with the specifications of the linked object. The default library is the current user working library.

12. Export an External Symbol

[exname:] EXPORT name

The 'EXPORT' directive is used to make an externally defined symbol,'name', visible in the relocatable object output. If 'exname' is specified it is used as the external name. Note. Any unreferenced external symbols always remain visible in the relocatable object.

13. End of Commands

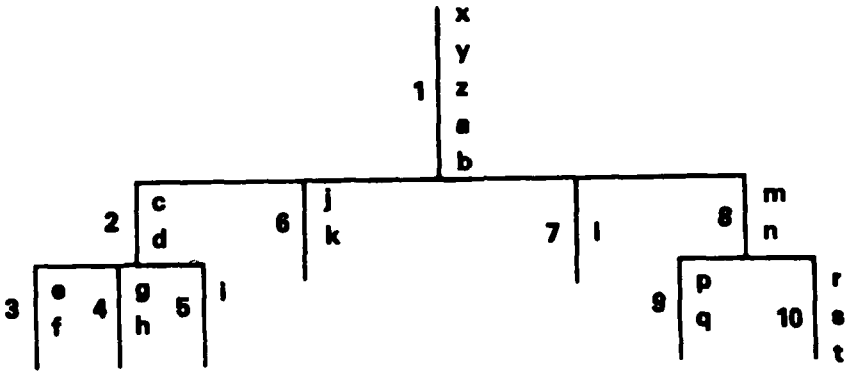
END

The 'END' directive terminates the set of Linker directives.

300

Example

The following diagram pictures a multi-level overlay structure for a program. Small letters are used to represent included objects; the segments are numbered.



301

The Link directives that produce this above program structures is shown below.

```
ONE       SEGMENT -- root
          INCLUDE x,y,z
          INCLUDE a,b
TWO       SEGMENT -- subordinate segment
          INCLUDE c,d
THREE     SEGMENT
          INCLUDE e
          INCLUDE f
FOUR      SEGMENT THREE -- overlay segment
          INCLUDE g,h
FIVE      SEGMENT THREE -- another overlay segment
          INCLUDE i
SIX       SEGMENT TWO -- overlay at level 2
          INCLUDE j,k
SEVEN     SEGMENT TWO -- overlay segments 2 and 6
          INCLUDE v
EIGHT     SEGMENT TWO -- overlay segments 2, 6 and 7
          INCLUDE m,n
NINE      SEGMENT -- subordinate to eight
          INCLUDE p,q
TEN       SEGMENT NINE -- overlay-segment 9
          INCLUDE r,s,t
          HEAP 999 -- heap and stack
          END
```


302

A summary of the Linker directives is given below with examples, any directive may be abbreviated.

```
A:     SEGMENT                --establish segment called "A"
       INCLUDE  X,Y           --include Relocatable objects X, Y
ALFA:  EQUATE  BETA           --equate ALFA to BETA
BUFF:  RESERVE 500            --reserve data area and name it BUFF
       ALIGN   2              --double word align the heap space below
HHH:   HEAP    1500           --reserve heap space and name it HHH
       ORIGIN  8192           --establish origin for stack
SSS:   STACK   300            --reserve stack space and name it SSS
       ENTRY   MAIN           --define entry point
       LIBRARY /SITE_DEF      --establish program library
EP13:  LIMIT   S1,S2,S3       --establish end of segments
       EXPORT  ADUMP              --make symbol 'ADUMP' visible in
                                  relocatable --object
```

APPENDIX B - PROGRAM INITIATOR

When a program is linked with the Load module option , the Linker inserts an "Initiator Routine" and a Segment Load Table at the beginning of the root segment of the program. When a program is invoked through the ACLI, the root segment is loaded and control is passed to the Initiator Routine.

The Initiator first performs any initialization required to obtain code and data work space for the program being executed. The Initiator uses the segment load routine, SEG_LOAD, to load and elaborate the segment containing the main program entry point and all of its parent segments.

The Initiator obtains the parameters specified at the program invocation and, using the main program specifications contained in the unit-specifications of the load object, verifies the parameters, establishes any default values and calls the main program entry point as a procedure.

The Initiator routine thus acts as the parent procedure for the main program. The Initiator routine also acts as the handler of last resort for all exceptions and on option will cause a post mortem memory dump for an unhandled exception.

When the main program exits, it will return to the Initiator routine. The Initiator routine will wait for any dependent tasks of the process to terminate and will then return to the ACLI passing back the return code supplied by the main program.

An unhandled exception will cause the name of the exception to be treated as the return code. In absence of a function main program or an unhandled exception the value 'ok' will be returned.

APPENDIX C - LOADING

Segment Loading

The dynamic loading of overlay segments at execution time will be performed through a segment load routine, SEG_LOAD, in conjunction with the segment load table. The segment load table is built by the linker and is contained in the root segment of a load object.

The Linker replaces calls on external procedures in a subordinate overlay segment with an indirect call through a load packet to the system routine SEG_LOAD: the load packet specifies the segment number and location of the procedure being called.

If the segment being referenced is not loaded, it and any unloaded parent segments in the program structure will be loaded and elaborated. Any segments which were overlaid in this process will be marked in the segment load table as unloaded and their load packets will be modified appropriately. After the segment being referenced is loaded, SEG_LOAD will modify the packet to pass control to the specified procedure directly.

After loading and elaborating the required segments, SEG_LOAD will pass control to the called procedure with the normal procedure linkage. SEG_LOAD is also used by the Initiator routine to load and elaborate the main program segment and any of its parent segments at program initialization.

Segment Load Table

The segment load table is used for the dynamic loading of overlay segments at execution time. The table describes the overlay structure of the program and for each segment the following information is held:

1. the load address of each location counter

2. the disc location (or identity) of the object text of the segment in the load object

3. the elaboration procedure address, this procedure will elaborate all library units in the segment in the proper order

4. a pointer to its parent segment and a pointer to a chain of its immediately subordinate segments.

A description of the segment load table record follows.

```
type SLT_PTR;
type SEGMENT_TABLE_ENTRY (NO_LOC_COUNTERS : INTEGER) is
    record
        SEGMENT_NO          :  INTEGER;  -- internal ordinal number
        SEG_NAME            :  SEGMENT_NAME;
        SEG_LOADED          :  BOOLEAN; -- true if segment is loaded
        SEG_RESIDENT        :  BOOLEAN;  -- true if segment is resident
        SEG_LEVEL           :  INTEGER; -- internal level number 0,1 2
        SEG_PARENT          :  SLT_PTR; -- parent segment
        SEG_SUB_CHAIN       :  SLT_PTR; -- subordinate overlay chain
        SEG_SIB_LINK        :  SLT_PTR; -- sibling segment
        DISC_TEXT_ADDR      :  DISC_ADDR; -- object text location
        DISC_DEBUG_ADDR     :  DISC_ADDR; -- debug table location
        LOAD_ADDR_ARRAY     :  array (0..NO_LOC_COUNTERS) of LOAD_ADDR;
        ELAB_PROCEDURE      :  EA_PTR; -- elaboration prologue address
    end record;

type SLT_PTR is access SEGMENT_TABLE_ENTRY;
```

A segment load control table is used with the segment load table to control segment loading.

```
type SEG_CONTROL_TABLE is
    record
        MAIN_SEG            :  INTEGER; -- segment number of main
                                        -- program
        CURRENT_SEG         :  INTEGER; -- lowest level loaded segment
```

```
        ENTRY_POINT          : EA_PTR; -- entry point address
    end record;

-- Segment Link Packet

type SEG_LINK_PACKET is
    record
        SLP_IDA              :  EA_PTR; -- indirect destination address
        SLP_CALLER_SEG       :  INTEGER; -- caller Segment number
        SLP_CALLEE_ADDR      :  EA_PTR; -- callee Address
        SLP_CALLEE_SEG       :  INTEGER; -- Callee Segment number
    end record;
```

Volume 7

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

MAPSE Editor

Prepared for

Rome Air Development Center

Griffis Air Force Base, NY 13441

Contract No. F30602-80-C-0292

309

TABLE OF CONTENTS

311

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This specification establishes the performance, design, development, and test requirements for the MAPSE Tool Set member, the Text Editor (EDIT).

1.2 FUNCTIONAL SUMMARY

The purpose of this specification is to define the Text Editor being designed as part of the Ada Integrated Environment contract for RADC. This document will serve to communicate the functional design decisions that have been adopted and to provide a basis for the continuing design effort.

The text Editor provides facilities for the creation and modification of text objects in the KAPSE Data Base (KDB). The capabilities provided include line, string and screen oriented text modification commands, input and output commands, macros and the ability to execute APSE command language interpreter (ACLI) commands.

The Editor has interfaces with the KAPSE Data Base System (KDBS), the CLI and the KAPSE Framework (KFW). The Editor has an indirect interface with all programs and tools which use text objects from the KDB, since the Editor is the primary means of creating and modifying these objects.

312

SECTION 2 - APPLICABLE DOCUMENTS

The following documents form a part of this specification to the extent specified herein.

2.1 PROGRAM DEFINITION DOCUMENTS

1. Reference Manual for the Ada Programming Language, July 1980
2. Requirements for Ada Programming Support Environment, "STONEMAN", February, 1980
3. Statement of Work, Contract No. F30602-80-C-0292, 80 Mar 26

2.2 INTER SUBSYSTEM SPECIFICATIONS

4. System Specification for the Ada Integrated Environment.
5. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.
6. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base.
7. Volume 3, Computer Program Development Specification for CPCI MAPSE Command Language Interpreter.
8. Volume 4, Computer Program Development Specification for CPCI MAPSE Configuration Management System.
9. Volume 5, Computer Program Development Specification for CPCI MAPSE Compiler.
10. Volume 6, Computer Program Development Specification for CPCI MAPSE Linker.
11. Volume 8, Computer Program Development Specification for CPCI MAPSE Debugger.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

13. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.
14. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

15. SOS Reference Manual
16. TECO Reference Manual (in DECSYSTEM10 User's Manual)
17. TECO (in Multics Programmer's Manual)
18. UNIX Editor
19. WYLBUR Reference Manual
20. Edm (in Multics Programmer's Manual)
21. Wordstar User's Guide, MicroPro International Corp.
22. UCSD Pascal System Reference Manual
23. CSTS GPS Reference, Vol. 1: General
24. GCOS
25. Inforex Text Editor Reference Manual.

314

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section provides the general description, identifies the external and internal interfaces, gives the functional requirements and presents the characteristics of the Configuration Item identified as the Text Editor (hereafter referred to simply as the Editor).

3.1.1 General Description

The purpose of the Editor is to allow the creation and modification of text objects which can be used as inputs to the APSE Command Language Interpreter, the Ada compiler, other tools and user programs. The Editor will be reentrant and sharable.

3.1.2 Peripheral Equipment Identification

The Editor is to operate on the IBM VM/370 and on the Interdata 8/32 under the OS/32 Operating System. The Editor will accept commands and text objects and produce listings and text objects in a device independent fashion.

The Editor will be sensitive to some of the characteristics of the command input and listing output devices. However, whether the input is echoed as full or half duplex and whether individual characters are obtainable will affect the behavior of the Editor. In addition, the Editor must know if it is being invoked interactively or through a command file.

The design will permit the editing of any size KDB text file in at most 32K bytes of memory.

3.1.3 Interface Identification

The text Editor will operate as an executable program under the MAPSE. It may be invoked by the ACLI in response to a user command, or by a call from an APSE tool or program.

The Editor will interface with the KDBS to create and access text objects, and with the ACLI to allow ACLI commands to be executed. The Editor will interface with the Configuration Manager for version control. There will be an interface to the user, interactively or through a command file.

3.1.4 Function Identification

The major functions of the Editor are:

1. Editor initialization
2. Command executor
3. Command parser
4. String handler
5. Expression evaluator
6. Environment maintainer
7. Intra-line Editor

3.2 FUNCTIONAL DESCRIPTION

The purpose of this section is to provide a detailed description of the major functions of the Editor and its relationship to other programs in the APSE.

3.2.1 Equipment Description

The Editor has no host or target machine dependencies.

3.2.2 Computer Input/Output Utilization

Inasmuch as the APSE (and hence the Editor) will be run on large host computers, the possibility of editing from dial-up terminals cannot be ignored. Thus, while the Editor will support screen-oriented editing, it cannot assume that all terminals which will be used will run at high speeds, and must support editing from slower terminals also.

The Editor will be able to operate in both half-duplex and full-duplex modes, but the interfaces for certain operations, notably intra-line editing, will not be as convenient in half-duplex mode.

File Interfaces

std_in - is the file from which the Editor commands are obtained
std_out - is the file to which listing and verification is done
std_err - is the file to which diagnostic messages are written
EDIT_START_UP - is the file which contains the Editor commands which are read by the Editor initializer

3.2.3 Computer Interface Block Diagram

See Figure 3-1.

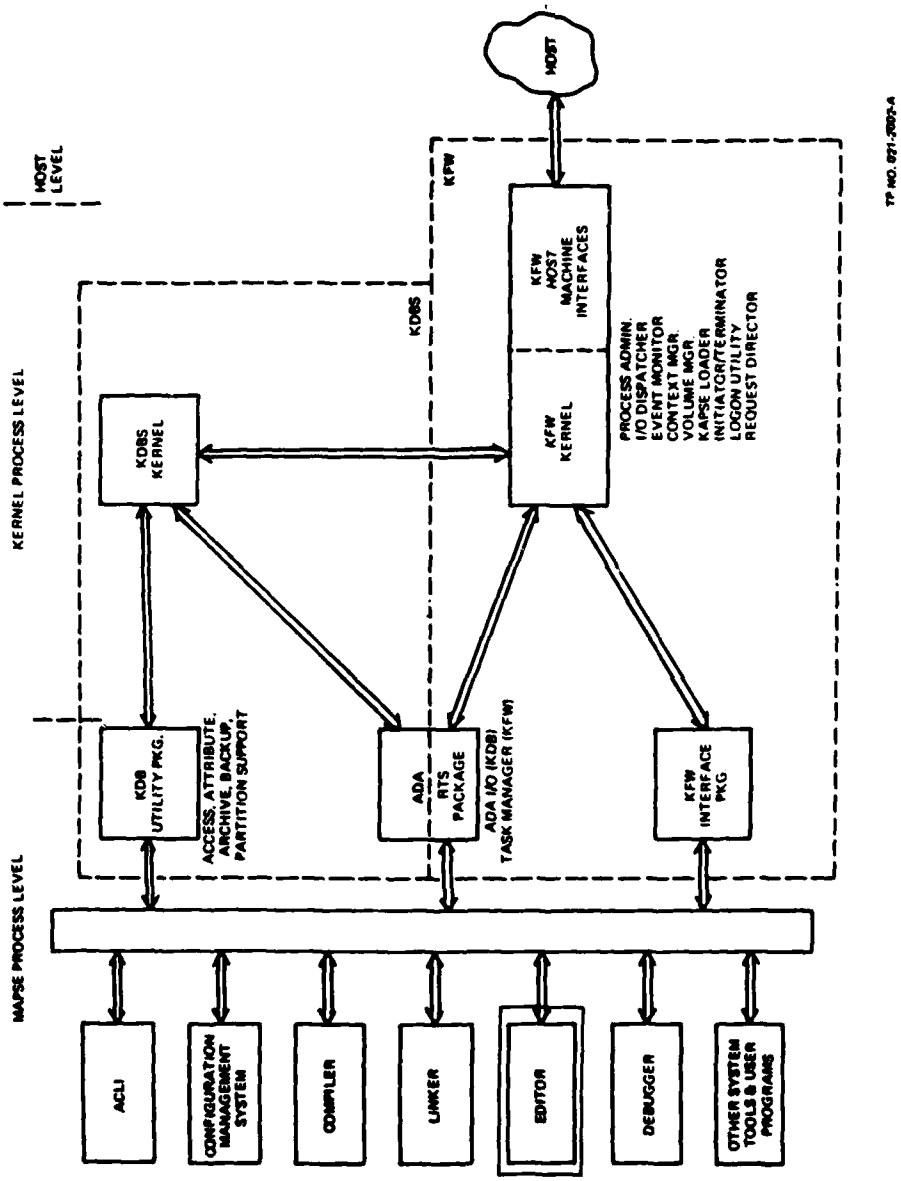


Figure 3-1. Interface Diagram

317

3.2.4 Program Interfaces

3.2.4.1 KAPSE Data Base

1. Read commands
2. Create text object
3. Open text object
4. Read a block of a text object
5. Write a block of a text object
6. The Editor will format blocks, so it will need to be aware of the block format used by the KDB
7. Obtain user commands from a command file or terminal

3.2.4.2 KAPSE Framework (KFW)

1. Query whether input is half-duplex or full-duplex
2. Turn off echoing, if full-duplex
3. Query whether input is obtainable character by character
4. Query whether the Editor was invoked interactively
5. Be informed if the user has entered an attention to interrupt the execution of a command

3.2.4.3 APSE Command Language Interpreter

1. Execute command

3.2.4.4 User

1. Commands.

3.2.4.5 Impositions on the other CPCIs

These are the interfaces which may impose additional requirements on other CPCIs.

KAPSE Data Base

1. Read a block containing a particular key.
2. Read the block containing the nth record before or after a given record.
3. It must be possible to efficiently access blocks in text files in a random order.
4. A system convention for entering upper/lower-case characters on single case terminals is assumed.
5. A system convention for tabbing is assumed.
6. In addition, the following special characters are required: carriage return, line feed, back space.

3.2.4.6 Editor Syntax Summary

The Editor may be operated in either interactive or batch mode. While the Editor commands are oriented toward interactive editing, all of the commands are available in either mode. The syntax of Editor commands is described below. The semantics of the commands are described in section 3.3.

Lower case names represent non-terminal symbols. Upper case letters and special characters (with the exception of {, }, }+, and +/- and blank) are *terminal symbols*. A syntactic entity enclosed in braces ({}) may occur 0 or more times. An entity enclosed in square brackets is optional.

Alternatives are given on separate lines with one alternative left-justified under the other. Continuation lines are set back from the alternatives. The symbol +/- stands for either a plus or minus sign.

In the syntax descriptions shown below the characters / and % are used for the string terminator and escape character, respectively. It will be possible for the user to define which characters are used. However, it was felt that the syntax is clearer when the default characters are used rather than symbols such as <string-terminator> and <escape-character>.

Commands

Name		Syntax	
Alter		A	rangelist
Buffer		B	<single-character>
Copy	[c]	C	[rangelist] [@ pos]
Delete	[c]	D	[rangelist]
Environment		E	<env-code> value
Find	[c]	F	string [rangelist]
Insert	[c]	I	string
KAPSE command		K	string
Locate line		L	[line-position]
Move	[c]	M	[rangelist] [@ pos]
Number		N	range [,number-1] [,number-2]
Print	[c]	P	[range list]
Q		Q	[file]
Read		R	file [rangelist][@pos]
Substitute	[c]	S	string string [rangelist]

V		V	verify
Write		W	file [rangelist]
eXecute command macro	[integer]	X	alphanumeric { string }
Zone		Z	zone definition

Note: Many of the command parameters are optional. The prose descriptions of the effects of the commands specify which these are.

Syntactic Components

String	{characters|buffer|context-string}/	
Context-string	[!b] char-position .. char-position	
Char-position	%A string %B string %# integer line-position [# integer]	
File	APSE-file-name /	
Buffer	! char	any single character from an implementation defined character set including at least the Ada character set
Position	line-key	(line-position)
	line-key # integer	(char-position)
	%A string	(char-position
	%B string	(char-position

321

Range	[!b] line-range
	[!b] char-range
Line-range	line-position [.. line-position]
	line-position $ integer
Char-range	char-position .. char-position
	line-position # char-position

Note: In a string "characters" may be any sequence of characters except the string terminator and the escape character. Escape sequences are listed below.

Integers may be replaced by integer expressions enclosed in parentheses.

Special Characters. Special characters may be specified by the user.

/	String terminator (default
%	Escape character (default)
$	Line count indicator
#	Character number indicator
@	workbuffer position indicator
..	Range indicator
!	Buffer indicator

Shorthand Notations (in this section braces are not metasymbols)

<	Beginning of file
>	End of file
{	Start of cursor (position)
}	End of cursor (position)
{}	Cursor (range)

Escape Sequences

%A	After
%B	Before
%Hdd	Hexadecimal character representation
%Oddd	Octal character representation
%%	% <escape>
%/	/ <string-terminator>
%!b	String contained in Buffer 'b' 'b' is any single character of an implementation defined character set which will include at least the ADA character set.
%*	Any number of any character including none
%?	Single wild card character
%-	Most recent deleted or found string
%R	Read string from std_in
%W	Write string to std_out
%+	Most recent substituted or inserted string
%L	Current line number
%<carriage-return>	Line continuation

Expressions

The Editor provides integer and boolean expressions.

Arithmetic operators

+	Addition
-	Subtraction, negation
*	Multiplication
/	Division

Boolean operators

&	And
!	Or
'	Not (prefix)

Relational operators

>	Greater than
=	Equal
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
/=	Not equal

324

Command Groupings

Sequential	command {; command}
Repetition	[count] command
Loop	[count] (command-list)
Case	[condition ¦ command-list { ¦ command-list } ¦¦ command-list]

Note: Command Termination

A semi-colon may be used to terminate any command. Semi-colons are only necessary to separate two commands on a line if the first command has an optional trailing parameter omitted and the second command has an optional leading parameter (a repetition count) omitted.

Carriage Return

Within a string, the carriage return is a part of the string. At the Editor command level, a carriage return ends the command.

Line Continuation

The sequence "%<carriage-return>" can be used to continue a line, the sequence would neither end a command nor be made part of a string.

3.2.4.7 Screen Oriented Capabilities

Although all editing functions are supported for both line and screen devices, some functions, such as cursor positioning, would be of little value on hard copy terminals. The following capabilities are oriented to high speed screen devices.

1.	Screen Size	The screen size is specified as part of the user environment.
2.	Positioning	Cursor positioning is supported, the user may specify the cursor positioning characters.
3.	Paging	Paging is supported with the P command. On a line oriented device P displays the next line, on a screen device P displays the next page.
4.	Exchanging	The exchange command will allow in-place text modification.
5.	Highlighting	Text highlighting through cursor positioning is supported. The highlighted text may be referenced with the Copy, Move, and Delete commands.

3.2.5 Function Description

The purpose of the Editor is to allow text objects to be created and modified. To this end, a number of commands are provided. These include: inserting, deleting, finding and replacing, copying and moving strings; inserting, deleting, copying and moving lines; cursor positioning; printing; renumbering; intraline editing (alter); reading and writing text objects; and defining and executing command procedures.

In addition, there are facilities for computing the results of expressions, and for executing commands conditionally and repetitively. It is also possible to execute ACLI commands from the Editor.

The main functions of the Editor are: initialization, command parsing, expression evaluation, string handling, command execution, intra-line editing, and environment maintenance.

3.2.5.1 Initialization

The initialization function sets up data which must be initialized, checks the arguments with which the Editor was invoked, establishes defaults and reads the user-defined Editor initialization file.

3.2.5.2 Command Parsing

The command parsing function is the main controller in the Editor. It is responsible for parsing commands and for calling the expression evaluator and command executor, intra-line Editor and environment maintainer, according to the type of command entered.

3.2.5.3 Expression Evaluation

The expression evaluator computes values for the arithmetic, boolean and relational expressions which appear in commands.

The string handling function is responsible for string matching and handling the special escape sequences which appear in strings.

3.2.5.4 Command Execution

Command execution is responsible for performing the actions specified by the commands. For clarity, each command is described in a separate subsection in section 3.3. The Editor commands included in command execution are: copy, delete, find, insert, execute ACLI command, print, move, number, procedure definition, quit, read, substitute, write and execute macro.

3.2.5.5 Intra-line Editing

Intra-line editing implements the alter command. It has the responsibility for parsing the special alter commands, and for modifying the text according to those instructions.

3.2.5.6 Environment Maintenance

The environment maintainer provides facilities for changing and querying the environment.

3.3 DETAILED FUNCTIONAL REQUIREMENTS

This section contains detailed descriptions for the major functions of the text Editor. In addition, the effect of each Editor command is explained. Examples of sample edit commands are presented in Attachment A.

There are a number of Editor requirements which relate to the Editor as a whole, rather than to a particular function. These are described here.

The Editor will not modify any input object unless it is given an explicit instruction to do so (and it is permitted to do so by the configuration manager).

The user will be protected against the accidental loss of data in the work buffer insofar as is possible. This includes such measures as requesting confirmation for massive deletions.

There are a number of general concepts which relate to the Editor as a whole which must be detailed for the descriptions of the functions to be coherent.Positions

The Editor will implement both character and line oriented commands. Positions may refer to characters or lines. A position which specifies a character position will be referred to as a "character-position". Similarly, a position which refers to a line will be called a "line-position".

The syntax for a position is:

line-key

or

line-key # integer

or

%A string

or

%B string

The line-key may be either a line-key from the work buffer, or one of the cursor abbreviations, { or }.

328

The cursor symbols give a line or character position. "{ # 1" converts a line cursor to a character cursor.

The third and fourth forms shown above yield the position immediately after and before the next occurrence of the sting.

A range consists of the characters or lines between two positions. A range which is specified by two line pointers is a "line-range". Any other range (i.e., one which has at least one character position is a "character-range". The cursor sumbol {} may be used to specify a range. The type of range is the same as the cursor type.

The syntax for ranges is:

```
[#b] position [ ..  position ]
or
[#b] line-position [ ..  line position ]  @
                char-position  ..  char-position
```

The first form gives a sequence of characters from the first position to the second. The second form gives a "rectangle" of characters between the two line-positions vertically, and the two character positions horizontally. If a buffer specification precedes the range, the range is within the specified buffer.

A carriage return is assumed to exist as the last character of every line. The line key is not considered part of the line. Thus, advancing two characters from the last character on one line will skip over the carriage return to the first character on the next line.

As part of its environment the Editor will maintain a cursor, which, in some sense, tells "where the Editor is". The normal application for this cursor is to provide a context for an operation if one is not specified in the command. For example, the insert instruction will insert text after the position specified by the cursor if a position is not given explicitly in the command.

The cursor is a "wide" one. It may point to a character, a line or a range of characters or lines. The cursor may be considered to be a range. Thus, it makes sense to talk about the beginning of the cursor and the end of a cursor. These are used in the descriptions of the Editor functions as shorthand notations for "the position of the start of the cursor" and the "position of the end of the cursor."

Another concept is that of a "buffer." Buffers may be used to hold values, text, strings and Macros. They may be read, written or edited. Buffers may be very small (a null string) or very large (an entire file). The work buffer is, in fact, a distinguished buffer. At any given time only one buffer (the work buffer) may be edited, although any of the existing buffers may be read or written. For each buffer containing text, the size, distinguished Zone, and cursor is maintained. The size of an empty buffer is zero. For each buffer containing a value, the value is maintained.

An additional concept is that of a macro of text Editor commands. The Editor will permit the user to define macros, to invoke them, and to pass them parameters.

Line numbers are optional. If line numbers are not used line number n will refer to the nth line in the file.

3.3.1 Editor Initialization

Initialization performs the housekeeping necessary to begin editing and reads the user-defined Editor initialization file.

3.3.1.1 Inputs

Initialization object - a KDB text object in the working directory with the name EDIT_START_UP. This file contains Editor commands (see 3.2.4).

Parameter list - the parameter list passed to the Editor by the CLI.

3.3.1.2 Processing

The initialization function is responsible for performing any housekeeping necessary to begin the execution of the Editor. This includes initializing tables, establishing defaults, and processing the parameter list which is passed to the Editor upon its invocation by the CLI.

The initialization function is also charged with ascertaining whether an EDIT_START_UP exists, and if so, opening the object and performing the initialization directives. The work buffer is initialized (at least in a logical sense) to the contents of the file(s) specified when the Editor was invoked.

3.3.1.3 Outputs

Editor internal tables - initialized.

3.3.1.4 Special Requirements

The object name EDIT_START_UP should not be used anywhere in the APSE for anything but Editor initialization files.

3.3.2 Command Parser

The command parser reads command lines, parses them into tokens, and diagnoses syntax errors.

3.3.2.1 Inputs

EDIT_START_UP - If an initialization file is present commands are read from it, prior to reading std_in.

std_in - The primary inputs to the command parser are Editor commands which are obtained as input from the user (directly or from a command file) or from one of the Editor's buffers. The syntax for the input is given in section 3.2.4.
The syntax of each command received is checked by the parser to insure that it conforms with the description above.

3.3.2.2 Processing

When the Editor is first invoked, the command parser requests the Editor initialization function to ready the Editor for execution.

The command parser obtains commands, either a line at a time or a character at a time, depending on the interface to the user and the limitations of the host operating system. Commands are obtained either from a text object (such as the initialization file), as the result of input from the user, or from one of the Editors buffers. Obtaining a command from a buffer may or may not involve input.

The command is checked for syntactical correctness and broken into tokens. Any expressions present in the command are evaluated by the expression evaluation function. Defaults are supplied where necessary and the values associated with shorthand operators are provided. The tokens are assembled into a canonical form and are passed to the appropriate Editor routine.

Syntax Checking

Each command is checked to insure that it is syntactically correct with respect to the Editor command language described in section 3.2.4. If an error is detected, a diagnostic is issued to the error output file, nested procedures and their parameters are unstacked and the command line at the top level is flushed.

The parser will perform a limited amount of semantic checking. This includes verifying that the lower limit of a range does, in fact, precede the upper limit, and that line keys and character positions are non-negative.

3.3.2.3 Outputs

std_err - diagnostics, if any.

Canonical command - the internal representation for the Editor command.

3'37

3.3.3 Expression Evaluation

The expression evaluator is responsible for computing the results for the expressions which appear in commands.

3.3.3.1 Inputs

Expression - from command line.

3.3.3.2 Processing

The expression evaluator parses and evaluates the expressions which occur in Editor commands. The priority of operators is as follows:

*, / Highest
+, -, ' (unary)
+, - (binary)
<, =, >, >=, <=, /=
&, | Lowest

The arithmetic operators (+, -, *, and /) take integer operands and yield integer results. The boolean operators (&, | and ') take boolean operands and yield boolean results. The relational operators (<, =, >, >=, <=, and /=) take boolean, integer or string operands and yield boolean results. If the operands of a relational are mixed, booleans are converted to integers, and integers to strings as necessary.

These operators could be overloaded with other types in an APSE Editor, but only the operations described above will be supported in the MAPSE Editor.

3.3.3.3 Outputs

Expression result - integer.

3.3.4 String Handling

The string handling function is responsible for performing string matching and processing the escape sequences which may appear in strings.

3.3.4.1 Inputs

One or more strings.

3.3.4.2 Processing

The string handler's primary responsibilities are string matching and processing the escape seuences which appear in strings.

String matching may be performed in one of two modes: character and token. Character mode matches strings character for character, the only exceptions being the escape sequences listed below.

Token mode treats the string as a sequence of Ada tokens. For a match to be successful, there must be the same sequence of tokens in the workfile. Note, in particular, that a token whose name appears in another token will not match that token. For example, "A" will not match the "A" in "AB" in token mode. White space (blanks, tabs, carriage returns) is insignificant except to delimit tokens. Thus, "A B" matches "A B", but not "AB". However, "A (" matches "A (", and also "A(".

There will be an option to permit the matching of strings, regardless of differences in case. This will be controlled by a flag in the environment.

The special character constructs (<escape><character>) are described below.

%A Converts a string to a position. The meaning of %Astring is the character position immediately after the string. This sequence may only precede a string; it may never appear in the middle of one.

%B Converts a string to a position. The meaning of %Bstring is the character position immediately before the string. This sequence may only precede a string; it may never appear in the middle of one.

%!b Buffer b. When this sequence occurs in a string, the effect is as if the contents of buffer b had been written in the string in place of the %!b.

%Hdd Hexadecimal character representation. When this sequence occurs in a string, the effect is to insert the character whose hexadecimal representation is dd into the string in place of the Hdd. Note that dd represents two hexadecimal digits.

%- Most recent found string. The effect of this construct appearing in a string represents the most recently found or deleted string.

%+ Most recent substituted string. The effect of this construct appearing in a string represents the most recently substituted or inserted string.

%R Read string from std_in. The syntax of this read command is "%Rstring". This construct causes "string" to be written to std_out, it is then replaced by the string read from std_in.

%W Write string to std_out. The syntax of this write command is "%Wstring". This construct causes the "string" to be written to std_out.

%Oddd Octal character representation. When this sequence occurs in a string, the effect is to insert the character whose octal representation is ddd into the string in place of the Oddd. Note that ddd represents three octal digits.

%P character-position .. character-position /

This construct allows a string to be represented as a range. The string consists of all of the characters between and including the two positions. Note that this construct may only appear in a search string and not in a replacement string.

%C Character. Gives character-for-character matching when token mode is the default.

%T Token. Marks the string as one which is to be matched in terms of tokens. This construct may only appear in a search string.

%? Wildcard. Matches any character. This construct may only appear in a search string.

%% Inserts the escape character in the string.

%/ /. Inserts the string terminator character in the string.

%¦ Or. This construct allows a search string to match either the string which precedes it, or the string which follows. For example, A ¦B/ would match either an "A " or a "B". This construct may only appear in search string.

335

%!n Parameter number n. Inserts the value of the nth parameter in the string. This construct is only valid during the execution of a macro which has at least n parameters. Parameter number 0 (!0) gives the number of parameters passed.

%* *. Indefinite wildcard. This construct matches any number of any character, including none. This construct may only appear in a search string.

%L L. Current line number. This construct may only appear in an expression.

3.3.5 Command Executor

The command executor is the controlling program for the Editor. It calls the Editor initializer at the start of execution, and repeatedly invokes the command parsing and execution functions to perform the operations specified in the commands which are input to the Editor.

The functional requirements for the various Editor commands are described in separate subsections for clarity. They could, however, be though of as part of the command executor.

3.3.5.1 Inputs

Command - The command is in a canonicalized form (see 3.3.1.3) with expressions having been computed and defaults having been supplied. The command is passed in by the command parser as a series of tokens.

Text objects - These are the text objects which were specified in the invocation of the Editor (either in an APSE command line or in a call on the Editor by some other tool or user program).

Buffers - Buffers contain values either as the result of the Editor initialization process or as the result of the execution of Editor commands.

3.3.5.2 Processing

The executor is responsible for executing statements sequentially, conditionally and repetitively. It also must determine which function is required by each command.

Upon completion of a command, the next command will be read, unless the last statement was in a conditional or a loop.

For the execution of a group of commands which has a repetition count or group range, tne commands are executed in order, until a command fails, the loop is exited, or until the termination criterion for the repeat count or the range is satisfied. The criterion for a repeat count of n is that all statements have been executed n times. The criterion for a range is that the cursor returned by any of the commands falls outside of the range. If either the repetition or the range is not specified, the termination criterion associated with that specification does not apply.

For the execution of a conditional command, the condition is first evaluated. If the value of the condition is n, the nth alternative (starting from 1) is selected. Note that boolean expressions return the value 0 for false and 1 for true. An alternative preceded by a double bar (||) is the else alternative. The else alternative is selected if the value of the condition is less than or equal to zero, or greater than or equal to the number of alternatives (i.e., if there is no alternative, not counting the else, which corresponds to the value of the conditional).

When an alternative is selected, the commands in it are executed. When the bar (|) at the end of the alternative is encountered, the remaining commands in the conditional group are skipped and the next command executed is the one which follows the conditional.

3.3.5.3 Outputs

Buffers - The contents of various buffers may be changed, depending on the command. The work buffer, in particular, is changed by a number of commands.

Environment - Various environmental parameters may be changed.

3.3.6 Alter Command A

The alter command allows intra-line editing.

3.3.6.1 Inputs

Command line - A range list

std_in - intra-line editing commands (see section 3.2.4).

3.3.6.2 Processing

The intra-line Editor allows changes to be made to text via a set of special alter commands. It behaves differently, depending on whether the interface between the user and the Editor is half-duplex or full-duplex. For this reason, this description is divided into two subsections.

Full Duplex

When the intra-line Editor is invoked, it positions the cursor to the first character position in the rangelist. The user may then issue any of the commands described below. Note that none of the intra-line commands are echoed. The only printing done is that described below.

Pointer moving commands

The pointer is never allowed to move outside the rangelist. If the pointer is at the right-hand end of a line in the rangelist, and an attempt is made to move right, the cursor moves to the leftmost character of the next line in the rangelist. An attempt to move right from the last character in the rangelist results in a wraparound to the first character in the rangelist. Moving left is the exact inverse of moving right.

A repeat count may be specified before any command in the list below. The effect of a repeat count of n (n>0) is the execute the command which follows, n times.

2,38

Moving down is analogous to moving right. Moving down causes the pointer to point to the character in the same position, but on the next line in the rangelist which has character which is in the same position but also in the rangelist. Note that the next line in the rangelist is not necessarily the next line in the file. If there is no line later in the rangelist which satisfies the above conditions, the pointer wraps around to the next character position and the top of the rangelist. If there is no character in the rangelist to the right or below the original pointer position, the pointer wraps around to the first position in the rangelist. Moving up is the inverse of moving down.

space Move one space to the right. Echo the character whose position was just left. If wraparound occurs, echo the key of the new line and all of the characters to the left of the current position.

backspace Move one space to the left. If wraparound occurs, echo the key of the new line and all of the characters to the left of the current position.

line feed Move down one line. Echo the remainder of the old line and the key and all of the characters to the left of the new pointer position of the new line.

caret Move up one line. Echo the remainder of the old line and the key and all of the characters to the left of the new pointer position of the new line.

return Move to the first character in the next line in the rangelist. Echo the remainder of the old line and the key and all of the characters to the left of the new pointer position on the new line.

Operators

B Bye. Leave intra-line mode. Changes made to the current line are retained.

D Delete. Delete the character pointed at. Deleted characters are echoed enclosed in square brackets.

5534

E Echo. Echo the remainder of the line, skip to a new line and echo the line key and the characters to the left of the pointer. The pointer position is unchanged.

F string Find. Find the next occurrence of the string in the rangelist. If the string is found, on the same line, echo the characters from the old pointer position to, but not including the new pointer position. If the string is found on a different line, echo the remainder of the old line, the key of the new line and all of the characters on the new line to the left of the new pointer position.

I string Insert. Insert the string characters to the left of the pointer. Echo the string characters but not the I and the string terminator. The pointer still points to the same character, but that character is now in a new position.

L Lower case. Change the pointed-to-character to lower case, if possible, echo the character, and move one position to the right.

Q Quit. Restore the pointed to line to its form before the current set of changes were made, and exit intra-line mode. Note that if a line is altered and the pointer is moved to another line, the changes in the first line are made permanent, and are unaffected by a Q. Only those changes made since the pointer was moved to the line are undone.

R string Replace. Replace the characters starting at the pointed to character with the characters in the string on a 1 for 1 basis, until all of the characters in the string are used up. Echo only the replacement characters and any unchanged characters to the left of the replacement characters if changes are made to more than one line.

S Slide. Slide the token pointed at to the right by deleting a blank to its right and inserting one to its left. If there is another token to the right, the command is ignored and a bell is transmitted to std_out.

U Upper case. Change the pointed at character to upper case, if possible. Echo the changed character and move one position to the right.

340

X Exchange characters. Switch the positions of the pointed to character and the character to its right. Move right one position and echo one character.

Z string Zap. Causes any characters that the pointer passes over to be deleted. Only cursor movement characters are allowed in the string.

Half Duplex, or Line at a Time

If the interface between the user and the Editor is half-duplex, or characters are transmitted a line at a time, intra-line editing is different from that described above. The concept of a pointer is different, and the effect of some of the commands is different. Commands are only executed after an entire line has been received.

Initially, the first line in the rangelist is printed. It is then possible for the user to enter intra-line commands.

Pointer moving commands

The pointer is never allowed to move outside the rangelist. If the pointer is at the last line in the rangelist and an attempt is made to move down, the pointer wraps around to the first line in the list.

Command	Action
caret	Move up one line. Print the previous line in the rangelist. Print spaces so that the next character typed will line up under the first character in the line which is in the rangelist.
return	Perform the actions specified by the commands on the line. If the last character on the line was a continuation character (-), print the modified version of the same line. Otherwise, move down one line and print the next line in the rangelist. Print spaces so that the next character typed will line up under the first character in the line which is in the rangelist.

Operators

Operators never take effect until the return is entered to terminate the line. In the descriptions below, reference is often made to "the character

241

directly above" some operator. This refers to the character which is printed in the same column as the operator, but on the previous line. Operators only take effect if they are applied to characters which are in the rangelist. (Characters which are outside the column range but in the same line are not affected).

Operator Action

space Serves only to position the other operators and operands.

B Bye. Leave intra-line mode. Changes made to the current line are retained.

D Delete. Delete the character immediately over the D.

I string Insert. Insert the characters in the string before the character directly over the I.

L Lower case. Change the character immediately over the L to lower case, if possible.

Q Quit. Ignore the changes specified on this intra-line command line and leave intra-line mode.

R string Replace. Replace the characters starting at the character directly above the R with the characters in the string on a 1 for 1 basis, until all of the characters in the string are used up.

U Upper case. Change the character directly above the U to upper case, if possible.

X Exchange characters. Switch the positions of the character directly above the X with the character on its right.

- Continue. When this character appears as the last character on the intra-line command line, it indicates that the carriage return which follows it is not to cause the pointer to move down a line.

3.3.6.3 Outputs

std_out - Echoing and printing as described above.

Work file - Changes to text in the rangelist as described above.

3.3.6.4 Special Requirements

In order for the full-duplex, character-at-a-time intra-line editing to be effective, it must be possible for characters to be obtained from the user one-at-a-time. In addition, it must be possible for the Editor to suppress the echoing.

While these commands are available for non-interactive use, their effectiveness in batch is likely to be greatly diminished, since they owe a good deal of their usefulness to the alignment between the printed representation of the text line and the cursor or print element on a terminal.

3.3.7 Buffer command B

The Buffer command causes the specified buffer to become the work buffer.

3.3.7.1 Inputs

Command line - B alphanumeric

Buffer - The buffer specified by the alphanumeric character

3.3.7.2 Processing

The Buffer command makes the specified buffer the new work buffer. The old work buffer is not lost, but is retained as an ordinary buffer. It may be made the current work buffer again through the use of another B command. The work buffer which is in use before the first B command is Buffer <blank>.

If the buffer has not been initialized, an empty work buffer is provided.

Verification: If the cursor associated with the buffer is defined, the line containing it is printed.

3.3.7.3 Outputs

std_out - verification as described above

Cursor - Has the same value as the cursor had the last time the buffer was the work buffer. If the buffer was not previously a work buffer, the cursor points to the beginning of a null work buffer.

3.3.8 Copy command C

The Copy command is used to move data to a new location in the work buffer without deleting the original.

3.3.8.1 Inputs

Command line - [count] C [rangelist][@position]

workbuffer - the current work buffer

3.3.8.2 Processing

The copy command causes the text specified in the rangelist to be copied to the current or specified position. If the position is a line position, the first line moved becomes the next line after that line. If the position is a character position, the first character of the moved text follows immediately after that position.

If any range specified in the rangelist overlaps the destination, the move is suppressed and a diagnostic is issued to std_err.

If new lines are created, either because line range(s) are specified, or because a character range includes an end of line, the first new line will be at (current-key + current-increment), with each subsequent copied line having a key current-increment higher than the previous. If this would cause line(s) to be out of order, the copy is suppressed and a diagnostic is issued.

The text specified in the rangelist remains unchanged.

Repeat count: If the repeat count is n, then n copies (total) of the text specified by rangelist are inserted at position. A negative count is illegal.

Defaults: If no rangelist is specified, the current cursor is used. If no position is specified, the end of the current cursor is used.

3.3.8.3 Outputs

Std_err - diagnostics, if any.

workbuffer - the work buffer now contains the data which was copied at its new (as well as old) location.

Cursor - the cursor points to the new copy of the data which was copied, unless the copy failed, in which case the cursor is unchanged.

245

3.3.9 Delete command D

The Delete command allows text to be deleted from the work buffer.

3.3.9.1 Inputs

Command line [±count] D rangelist

workbuffer.

3.3.9.2 Processing

The lines specified in the linerange are deleted from the work buffer. If none of the lines specified in the linerange exist, a warning diagnostic is issued.

Repeat Count: A repeat count is only allowed if no rangelist is specified. If the current cursor is a line-position, nD will delete n lines starting at the current line, -nD will delete the n lines preceding the current line. If the current cursor is a character-position, nD deletes all characters in the line to the right of the start of the cursor and the following n-1 lines, -nD deletes all characters in the line to the left of the cursor and the preceding n-1 lines.

Verification: The lines, if any, containing the characters immediately preceding and immediately following the deleted text are printed.

3.3.9.3 Outputs

Buffer !- - Buffer !- contains the deleted text.

workbuffer - lines are deleted as stated above.

std_err - diagnostic messages, if any.

Cursor - If lines were deleted, the cursor points to the beginning of the first line following the last line deleted. If the last line in the buffer was the last line deleted, the cursor left at the end of buffer.

3.3.10 Environment command E

The Environment command allows environmental data to be set and queried.

3.3.10.1 Inputs

Environment - see below.

Command line - E<code><value>

3.3.10.2 Processing

The environment maintainer allows editor environmental data to be changed or printed.

The Code must be the name of one of the environmental parameters in the list below. If the name is valid, the value is checked to see if it is a valid value for that particular parameter. If it is, the value is changed.

The following table gives the names of the environmental data, a description of legal values for each and its default value

Code	Name	Default	Value	Use
E		%	<escape-char>	Escape character
T		/	<term-char>	String Terminator
C		N	Y	Case Matching
			N	Yes or No
M		C	T	Ada Token Matching
			C	Literal Matching
V		Y	Y	Verify Yes or
			N	No

D	N	Y	Automatic InDenting
		N	Yes or No
I	10	Integer or Fixed Number	Line Increment
L	This code prints the current environment		

*The escape character may not be set to the string terminator or a digit. The string terminator may not be set to the escape character.

3.3.10.3 Outputs

std_err - diagnostic messages.

std_out - listing of the value of the requested environmental data.

Environment - value modified as stated above.

3.3.11 Find command F

The Find command locates a string in the work buffer.

3.3.11.1 Inputs

Command line - [±count] F string [rangelist]

3.3.11.2 Processing

The string is searched for in the text delimited by the rangelist. If the string is found, the cursor is set to point to the string.

Repeat count: If a repeat count of n is specified, the nth occurrence of the string in the text delimited by the rangelist is searched for. A negative count causes a backward search over the range.

Defaults: If no rangelist is specified, the search is from the next character or line position (depending on whether the cursor is set to a string or line position, respectively) to the end of the buffer, or to the beginning of the buffer for a reverse search.

If a null string is specified, the most recent search string is used.

Verification: If verification is on, the line(s) containing the search string are printed.

3.3.11.3 Outputs

std_err - diagnostic messages, if any.

Cursor - Points to the string if one was found.
Otherwise, it is unchanged.

Buffer !- is set to the string specified in the command. If the string was not found buffer !- is unchanged.

3.3.12 Insert command I

The Insert command allows text to be inserted in a buffer.

3.3.12.1 Inputs

Command line - {±count} I string.

workbuffer.

3.3.12.2 Processing

The text given in the string is inserted immediately after the current position.

If new lines must be created in the work buffer because because carriage return(s) appear in the string, the current increment will be used. The first new line to be inserted will have the key (current-key + current-increment). Subsequent lines will have keys each more than the previous key by the value of current-increment.

If the numbering scheme stated above would cause the sequence numbers to no longer be in strictly ascending order, the string will be saved as the current replacement string, but no change will be made to the work buffer. A diagnostic will be issued if this occurs.

Indenting: If auto-indenting is turned on, each time a new line is inserted, it will begin with the same number of blanks as there were leading blanks on the previous line. Any blanks entered at the start of the line will cause further indenting. Backspaces and back tabs may be used to undo indenting. On full-duplex terminals the appropriate number of blanks will be printed after the prompt. In half-duplex mode the supplied blanks will not be echoed, but will be entered into the buffer.

Repeat count: If the command is given a repeat count of n, the string will be inserted n times. A negative count inserts the text in front of the current position.

If a null string is specified, the most recent replacement string will be used.

3.3.12.3 Outputs

workbuffer - The string is inserted in the work buffer.

std_err - diagnostic, if any.

Cursor - The cursor points at the most recently inserted string.

Buffer !+ - Buffer !+ (replacement string) is set to the string specified in the I command.

3.3.13 KAPSE Command Processor K

The K command causes a command string to be passed to the ACLI.

3.3.13.1 Inputs

Command line - K string.

3.3.13.2 Processing

The string is passed to the ACLI to be interpreted as a KAPSE command. The ACLI is called as a subordinate task. Editing continues after the completion of the command.

3.3.13.3 Outputs

std_out - Diagnostic messages, if any.

KAPSE Command -The string from the Editor command line is passed to the ACLI.

3.3.14 Locate line command L

The Locate command positions the cursor to the specified line position.

3.3.14.1 Inputs

Command line - [±count] L [line-position]

workbuffer.

3.3.14.2 Processing

The Locate command causes the cursor to be set to the line position of the specified line.

Repeat Count: A repeat count is allowed only if no line position is specified, this default is described below.

Default: If no line position is specified, nL positions the cursor to the nth following line, -nl positions the cursor to the nth preceding line, oL positions the cursor to the current line.

3.3.14.3 Outputs

cursor - cursor set to specified line position.

std_err - diagnostic if specified line does not exist.

3.3.15 Move command M

The Move command causes text to be transferred from location(s) in the workbuffer to the current or specified position.

3.3.15.1 Inputs

Command line - [count] M [change rangelist][@position]

workbuffer.

3.3.15.2 Processing

The Move command causes text to be transferred from one (or several) locations in a buffer to the current or specified position. If the position is a line position, the first line moved becomes the next line after that line. If the position is a character position, the first character of the moved text follows immediately after that position.

If any range specified in the rangelist overlaps the destination, the move is suppressed and a diagnostic is issued to the current error file.

If new lines are created, either because line range(s) are specified, or because a character range includes an end of line, the first new line will be at (current-key + current-increment), with each subsequent moved line having a key current-increment higher than the previous. If this would cause line(s) to be out of order, the move is suppressed and a diagnostic is issued.

After the lines are moved, the lines specified by rangelist are deleted. This deletion does not occur if the move is suppressed.

Repeat count: If the repeat count is n, then n copies (total) of the text specified by rangelist are inserted at position.

Defaults: If no rangelist is specified, the current cursor is used.

If no position is specified the end of the current cursor is used.

3.3.15.3 Outputs

std_err - Diagnostic if position is within the range, or if the move would cause lines to be out of order.

workbuffer - The lines specified by rangelist are moved to position as specified above.

Cursor - The cusor points to the lines moved by this command.

3.3.16 Number command N

The Number command causes the workbuffer, or a portion thereof, to be renumbered.

3.3.16.1 Inputs

Command line - N[range][,number-1][,number-2]

workbuffer

3.3.16.2 Processing

The number command causes the workbuffer to be renumbered. The lines in the specified range are renumbered starting at number-1 with an increment of number-2. If the number command would cause any keys to be other than in ascending order, the numbering will be halted before this occurs, and a diagnostic will be issued to the current error file.

Repeat count: A repeat count has no effect on this command.

Defaults: If no range is specified, the entire buffer is renumbered. If number-2 (the increment) is not specified, the current increment will be used. If number -1 (the starting number) is not specified, the first line of the range will be used as the starting number.

3.3.16.3 Outputs

std_err - diagnostic if renumbering would cause lines to be out of order.

workbuffer - The lines which had old keys in the specified range are renumbered, starting at number-1 with an increment of number -2.

Cursor - set to point to the renumbered lines.

3.3.17 Print Command P

The Print command causes text to be printed on std_out.

3.3.17.1 Inputs

Command line - P [rangelist]

workbuffer

3.3.17.2 Processing

The Print command causes the text specified in the rangelist to be printed on the current output buffer. Line positions cause entire lines to be printed. Character positions cause the printing of only the specified text.

Repeat Count: A repeat count with a range list causes the text to be printed the specified number of times.

Default: If no rangelist is specified, nP causes the n lines following the current line to be printed, -nP causes the n lines preceding the current line to be printed.

3.3.18 Introduction -- Quit command Q

The quit command causes a termination of the Editor and a return to its invoker.

3.3.18.1 Inputs

std_in - confirmation of whether to quit if data may be lost.

3.3.18.2 Processing

If the work buffer has been modified since the last time it was written and the Editor was invoked in interactive mode, the user is warned that changes were made to the work buffer and asked to confirm the quit. If the reply is in the negative, then no further action is taken and the command executor resumes control. If a positive response is given, or if it was unnecessary to ask the question, the Editor is terminated.

As part of the termination process, any temporary buffers created by the Editor are deleted.

3.3.18.3 Outputs

std_out - Request for confirmation if data will be lost

Editor temp buffers - deleted

3.3.19 Read command R

The read command causes text to be read from a specified file into the work buffer.

3.3.19.1 Inputs

Command line - [count] R [file] [rangelist] [@ position]

Workbuffer

File specified in command line

3.3.19.2 Processing

The read command causes the lines specified in rangelist to be read from the file specified in the command line into the specified position in the work buffer. If for some reason the file cannot be read, a diagnostic is issued.

If new lines are created, either because line ranges(s) are specified, or because a character range includes an end of line, the first new line will be at (position-key + current-increment), with each subsequent moved line with a key current-increment higher than the previous. If this would cause line(s) to be out of order, the read is suppressed and a diagnostic is issued.

Repeat count: If a repeat count of n is specified, a total of n copies of the specified lines are inserted at the specified position.

Defaults: If no line range is specified, the entire file is read. If no file is specified, the current read file is used. If no position is specified, the end of the current cursor is used.

3.3.19.3 Outputs

std_err - if the read fails, a diagnostic is issued.

workbuffer - the lines read from the specified file have been copied to the specified position in the work buffer.

359

Current read file - the file name specified in the command is remembered as the current read file.

Cursor - If the read is successful, the cursor is set to point at the lines read by the command.

3.3.20 Substitute command S

The substitute command causes one string to be replaced by another.

3.3.20.1 Inputs

Command line - [count] S string-1 string-2 [rangelist]

workbuffer

3.3.20.2 Processing

The substitute command causes the specified occurences of string-1 to be replaced by string-2. If no repeat count is specified, but a range is, all occurrences within that range are replaced. If neither a range nor a repeat count is specified, only the next occurrence is changed.

Repeat count: If a repeat count of n is specified, the first n occurrences of string-1 within the range are replaced by string-2.

Defaults: If no repeat count is specified, string-2 is substituted for all occurrences of string-1 within the bounds of the rangelist. If no rangelist is specified, the range used is from the next character or line position (depending on whether the cursor is set to a string or line position, respectively) to the end of the buffer. If either string-1 or string-2 is a null string, the most recent search string or the most recent replacement string, respectively, is used.

3.3.20.3 Outputs

workbuffer - Occurrences of string-2 are substituted for string-1 as specified above.

Cursor - Points to the most recently substituted string-2, if at least one substitution was made. If no substitution was made the cursor position will be unchanged.

Search string - set to string-1 (Buffer !-)

Replacement string - set to string-2 (Buffer !+)

3.3.21 Write instruction W

3.3.21.1 Inputs

Command line - W [file] [rangelist]

Work file

3.3.21.2 Processing

The write command causes text to be written to a specified file. The workbuffer text described by rangelist is written to the file. Workbuffer keys are used for the output file records. If the file cannot be written for some reason, a diagnostic is issued.

The cursor position is unchanged.

Repeat count: A repeat count has no effect on this command.

Defaults: If rangelist is not specified, the entire workbuffer is written. If a file name is not specified, the current output file is used. Note that the current output file is initialized to the edited object when the Editor is entered.

3.3.21.3 Outputs

std_err - diagnostic if the file can't be written

File specified in command line - the text specified by the rangelist is written to the buffer.

Current output file name - If a file was specified in the command line, the current output file name is set to that file.

3.3.22 EXecute macro command X

The eXecute macro command cases an Editor command macro to be executed.

3.3.22.1 Inputs

Command line -- integer X character {string}

Buffer -- the contents of the buffer designated the character.

3.3.22.2 Processing

The number of strings following the command must be equal to the integer which precedes it. If this is not true, a diagnostic is issued.

The strings will be pushed onto a parameter stack. While the macro is being executed, the ith parameter will be available as %!i. Recursive execution of macros is permitted. If there is not enough room to save the parameters, a diagnostic will be issued.

The next command to be executed by the command executor will be taken from the buffer designated by the character specified in the command. Commands will continue to be taken from the buffer until the end of the buffer is encountered or an error occurs.

3.3.22.3 Outputs

std_err - diagnostic messages, if applicable.

next command pointer - points to buffer

3.3.23 Zone Mark Command Z

The Zone command is used to mark a specified range of characters as being distinguished. This distinguished range of characters may be used as the source string of a Copy, Move or Print command and may also be deleted with the Delete command. The Zone command uses cursor movements and string searches to identify the distinguished range of characters.

Note: Within a buffer only a single distinguished Zone is recognized. The Zone command causes an existing distinguished Zone to be no longer a distinguished Zone.

3.3.23.1 Inputs

Command line - Z {{<char>}|{cursor-move}}/

3.3.23.2 Processing

The Zone command causes a specified range of characters to be marked as distinguished. The cursor position at the start of the command is considered an anchor position. The Z command is directed to perform cursor movements through cursor movement characcters and string searches. At the completion of the Z command the cursor position is the final position and all characters between the anchor position and the final position are marked as being in a distinguished Zone.

Cursor Movements

- Move one space right
- Move one space left
- Move up one line
- Move down one line

String Searches

When one or more non-cursor movement characters are entered a forward search for the string is made and the cursor is positioned to the right of the string.

A cursor movement character ends a search string. A slash ends the Zone command.

3.3.23.3 Outputs

The specified range of characters are marked as being distinguished. The distinguished Zone may be referenced in a Copy, Move, Print or Delete command as C*, M*, P* or D*.

3.4 ADAPTATION

This section describes the the data requirements of the Editor with respect to system environment, system parameters and system capacities.

3.4.1 General Environment

On some systems the user interface will be limited to half-duplex mode. Low-speed lines would preclude the use of screen-oriented echoing.

3.4.2 System Parameters

The block size for text object may differ from system to system.

3.4.3 System Capacities

The number of buffers available on a given system may be limited by the number of objects which the MAPSE allows to be open on a given host. There may be a limitation on the total space available for buffers on some systems. The size of an object which may be edited is limited to the size of MAPSE text objects, which in turn may depend on the host.

3.5 CAPACITY

Not applicable.

SECTION 4. QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the MAPSE Editor. The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the Editor. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data are processed. For example, a review of hard copy test data might be used to verify that the values of specific parameters are correctly computed.

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

4.1.1 Subprogram Testing

Following unit testing, individual modules of the Editor shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2. Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

CPCI testing shall be performed on all development software of the Editor. This specification presents the performance criteria which the developed CPCI must satisfy. The correct performance of the Editor will be verified by testing its major functions. Successful completion of the program

testing that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3. System Integration Testing

System integration testing involves verification of the integration of the Editor with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the Editor performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

Subprogram testing will take two forms during the development of the Editor. Initially, scaffolding routines and debugging printouts will be necessary. Later, it will be possible to test the Editor through the use of Editor commands.

initial subprogram testing will concentrate on the command executor and the command parser, since these functions must be working in order for Editor commands to be executed. During the early stages of development, listings of the tokens created by the parser and lines read and written by the Editor will be used to verify that statements are being parsed properly.

Later on in the development, subroutines will be tested by entering commands that exercise those particular subroutines. The Editor will be designed so that there is a close relationship between the command entered and routines called, so it will be possible to check out a given routine by entering the appropriate commands.

In addition, the environment maintainer will provide flags which can be set by environment commands, and which may be checked so that debugging printouts can be produced.

Table 4-1. Test Requirements Matrix

SECTION	TITLE	INSP.	ANAL.	DEMO.	REV.	PARA. NO.
3.3.1	Editor Initialization	X			X	4.2.1,4.2.2
3.3.2	Command Parser	X			X	4.2.1,4.2.2
3.3.3	Expression Evaluation				X	4.2.2
3.3.4	String Handling	X			X	4.2.1,4.2.2
3.3.5	Command Executor				X	4.2.2
3.3.6	Alter Command				X	4.2.2
3.3.7	Buffer Command				X	4.2.2
3.3.8	Copy Command				X	4.2.2
3.3.9	Delete Command				X	4.2.2
3.3.10	Environment Command				X	4.2.2
3.3.11	Find Command				X	4.2.2
3.3.12	Insert Command				X	4.2.2
3.3.13	KAPSE Command Proc.				X	4.2.2
3.3.14	Locate Line Command				X	4.2.2
3.3.15	Move Command				X	4.2.2
3.3.16	Number Command				X	4.2.2
3.3.17	Print Command				X	4.2.2
3.3.18	Quit Command				X	4.2.2
3.3.19	Read Command				X	4.2.2
3.3.20	Substitute Command				X	4.2.2
3.3.21	Write Command				X	4.2.2
3.3.22	EXecute Macro Command				X	4.2.2
3.3.23	Zone Mark Command				X	4.2.2

3 11

4.2.1 Inspection

During development the Editor will be tested by creating and modifying test objects and then printing them using file dump and file tools. These tool will already be in the APSE toolset, so they will not exist just for Editor testing.

4.2.2 Review of Test Data

The testing of the Editor will use prepared scripts and command files and text objects which serve either as input to the Editor or for comparison with Editor ouput. The results of edit sessions will be compared to files which contain the correct results, by means of a file comparison tool.

One test will ve to create a CLI command file which can be executed. If it executes correctly, the interface is good. Another test will be to list an Editor created file with a listing utility. The reformated output from the complier will be used to verify that the Editor can, in fact, read a file created by another tool.

4.3 ACCEPTANCE TESTING

Acceptance testing shall involve comprehensive testing at the CPCI level and at the system level. The CPCI acceptance tests shall be defined to verify that the Editor satisfies its performance and design requirements as specified in this specification. System acceptance testing shall test that the MAPSE satisfies its functional requirements as stated in the System Specification.

Acceptance testing will consist of a more formal version of program testing. Prepared scripts and command files will be used. However, since the goal in acceptance testing is to insure a correct product, rather than to isolate problems, the granularity of the test may not be as for program testing. Again, a file comparison tool will be used.

The following area must be tested during acceptance testing:

Syntax checking - correct and incorrect commands

Incorrect commands must be diagnosed.

Commands

- All command types
- All argument types
- With and without defaults
- Cursor setting

Intra-line editing

- All intra-line commands

Conditionals

- Different values for condition
  - In range/out of range (with and without else)
- Failures within conditional group

Repeated groups

- With range, repeat count, both

Echo modes (half-/full-duplex)

Verification (on/off)

Matching (string/Ada token/text token)

String processing

- All excape sequences

Abbreviations

- Cursor, etc.

Create/modify files

Multiple file inputs

Environment

- Set
- List

Buffers

- Define procedures
- Execute procedures
  - Various numbers of parameters
- Switch work file
- Retrieve buffer value

These tests shall be conducted by the CSC/SEA team and formally witnessed by the government. Satisfactory performance of both CPCI and system acceptance tests shall result in the final delivery and acceptance of the MAPSE system. All formal testing will be witnessed by the Government.

SECTION 5. DOCUMENTATION

5.1 GENERAL

The documents that will be produced during the implementation phase in association with the test Editor development are:

1. Computer Program Development Specification
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. Users Manual
6. Rehostability Manual
7. Language Reference Handbook

5.1.1 *Computer Program Development Specification*

The final text Editor B5 Specification will be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II. A single document will be prepared for the Editor that defines the functional capabilities and interfaces. Any dependencies on the host system will be addressed in the document. Additionally, characteristics of potential host systems which have had impact on the B5 specification will be presented.

5.1.2 Computer Program Product Specification

A type C5 specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document will be used to specify the Editor design and development approach for implementing the B5 specification. This document will provide the detailed description which will be used as the baseline for any Engineering Change Proposals. A single C5 will be produced for the Editor with different sections addressing the dependencies of the two host computers.

375

5.1.3 Computer Program Listings

Listings will be delivered which are the result of the final compilation of the accepted Editor. Each compilation unit listing will contain the corresponding source, cross-reference and compilation summary. The source listing will contain the source lines from any INCLUDEd source objects.

5.1.4 Maintenance Manual

An Editor Maintenance Manual will be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the Editor to be easily maintained by other than the developer. The documentation will be structured to relate quickly to program source. The procedures required for debugging and correcting the Editor will be described and illustrated. Sample run streams for compiling Editor components, for relinking the Editor in parts or as a whole, and for installing new releases will be supplied. The data base will be fully documented with pictures of record layouts where appropriate and data algorithms explained.

The Maintenance Manual will be organized with a standard outline and separate parallel volumes will be delivered which address the tailoring of the Editor to a particular host system. Debugging aids which have been incorporated as an integral part of the Editor will be described and their use fully illustrated. Special attention will be given to the description of the maintnance mode operation of the Editor used to aid in the pinpointing of Editor problems.

5.1.5 Users Manual

A Users Manual shall be prepared in accordance with DI-M-30421 which will contain all information necessary for the operation of the Editor. Because of the virtual user interface presented by the CLI, a single manual is sufficient for all host computers. Information relevant to specific host systems (such as maximum file sizes, etc.) will be contained in appendices. Sample Editor listings will be included in the manual.

A complete list of all Editor diagnostic messages will be included with supplemental information chosen to assist the programmer in locating and correcting Editor command errors.

5.1.6 Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual will be prepared which describes step by step the procedures for rehosting the text Editor on a different computer. Tips will be provided wich will guide the developer module by module as to what may be used entirely or in part.

5.1.7 MAPSE Tools Reference Handbook

A MAPSE Tools Reference Handbook will be prepared which will contain syntax diagrams for all KAPSE command constructs. Other handy information such as number conversion and ASCII encoding tables, and host system parameters will be included. A summary of the Editor's command syntax and a brief description of the commands will be prepared as part of this handbook.

377

APPENDIX A - SAMPLE EDIT COMMANDS

Some of the basic Editor commands are shown below.

A.1 F Find

Find a string

FABC/	Find the the first occurrence of the string "ABC" searching from the current position to the end of the workbuffer.
FXYZ/100	Find the first occurrence of the string "XYZ" in line 100.
-FALPHA/	Find the string "ALPHA" searching backward from the current position to beginning of the workbuffer
FBETA/300..500	Find the string "BETA" searching from line 300 thru line 500 inclusively
3FGAMMA/600	Find the third occurrence of the string "GAMMA" in line 600.
F/	Find the most recent search string searching from the current position to the end of the workbuffer
F%!P/300..400	Search forward for the string contained in buffer P from line 300 to 400.

A.2 D Delete

Delete lines

D 100	Delete line 100
D100..200	Delete lines 100 thru 200 inclusively
D300#7	Delete character seven on line 300
D320#11..15	Delete characters eleven thru fifteen inclusively on line 320
D61,37,43	Delete lines 61,37 and 43
3D	Delete three lines starting at the current line
D*	Delete the distinguished Zone.

A.3 S Substitute

Substitute string-1 with string-2

SABC/DEFG/200	Substitute all occurrences of the string "ABC" in line 200 with "DEFG"
3SABC/DEF/300..500	Substitute the first 3 occurrences of the string "ABC" in lines 300 thru 500
SALPHA/BETA/	Substitute the first occurrence of the string "ALPHA" with the string "BETA", searching from the current position to the end of the workbuffer.

S//	Substitute the first occurrence of the most recent search string with the most recent replacement string, searching from the current position to the end of the workbuffer.
3S/GAMMA/	Substitute the first three occurrences of the most recent search string with "GAMMA", searching from the current position to the end of the workbuffer.
SDELTA/EPSILON/ 10,20,33	Substitute every occurrence of "DELTA" in line 10,20 and 33 with "EPSILON"

A.4 C Copy

Copy text to a position, the source text is not deleted.

C30..60	Copy lines 30 thru 60 inclusive to the current position.
3C80	Copy line 80 three times to the current position.
C10,30..33,20	Copy line 10, lines 30 thru 33 inclusive and line 20 to the current position.
C200..250@710	Copy lines 200 thru 250 inclusive to the position immediately after line 710. If line 710 does not exist the first line copied is given line number 710.
C*	Copy the distinguished zone to the current position.
C!P	Copy the contents of buffer P to the current position.

Note: The current-increment will be used for numbering the copied lines. The first line copied is given a line number current-increment higher than the line number preceding the position copied into. When copying into a nonexistent line number, the first line copied is given that line number.

A.5 M Move

Move text to a position, delete the source text.

M55	Move line 55 to the current position
M110..150	Move lines 110 thru 150 inclusive to the current position
M90..95@120	Move lines 90 thru 95 inclusive to the position following line 120 or if line 120 does not exist to that corresponding position
M10,333,111	Move lines 10,333 and 111 to the current position
3M90@400	Move line 90 three times to the line following line 400 or to the corresponding position if line 400 does not exist
M*	Move the distinguished zone to the current position
M!Q@910	Move the contents of buffer Q to the line following line 910 or to the corresponding position if line 910 does not exist

Note: The copied lines are deleted from the source. Incrementing is performed just as for copy.

A.6 I Insert

Insert text into the current position.

The following example shows inserting several lines after line 2230 (assume current-increment=5, indenting on)(<CR>stands for carriage return, user commands are underlined)

```
L2230I<CR>
2235        begin<CR>
2240        SUM:=0;<CR>
2245        for J in RAINBOW loop/<CR>
```

The line number prompts will be supplied by the Editor. The "/" ended the insert.

The following example shows the insertion of a string within a line (assume verify mode, user commands are underlined)

```
FX:/<CR>
1935  procedure INCR (X:  );
I in out INTEGER/<CR>
1935 procedure INCR(X: in out INTEGER);
```

A.7 N Number

Number a range of lines.

N	re-Number the whole workbuffer with current initial number and current increment
N100..200,110,5	re-Number lines 100 thru 200 inclusive, start numbering at 110 in increments of 5

N333,350	re-Number line 333 as 350
N,100,100	re-Number whole file from 100 in increments of 100

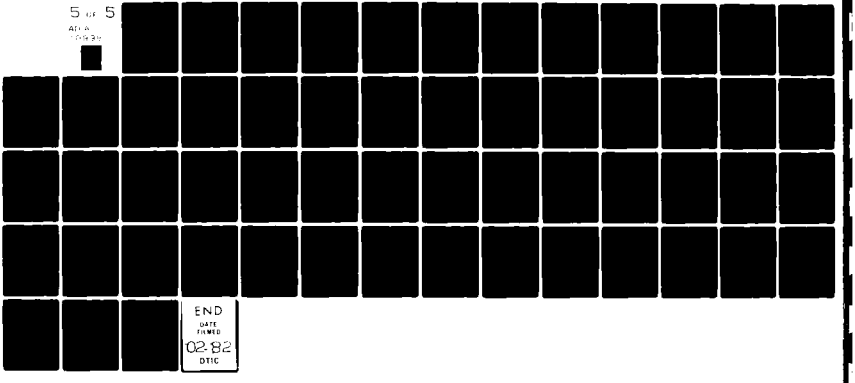
A.8 R Read

Read (a portion of) an Ada Object into the workbuffer, only text objects may be read.

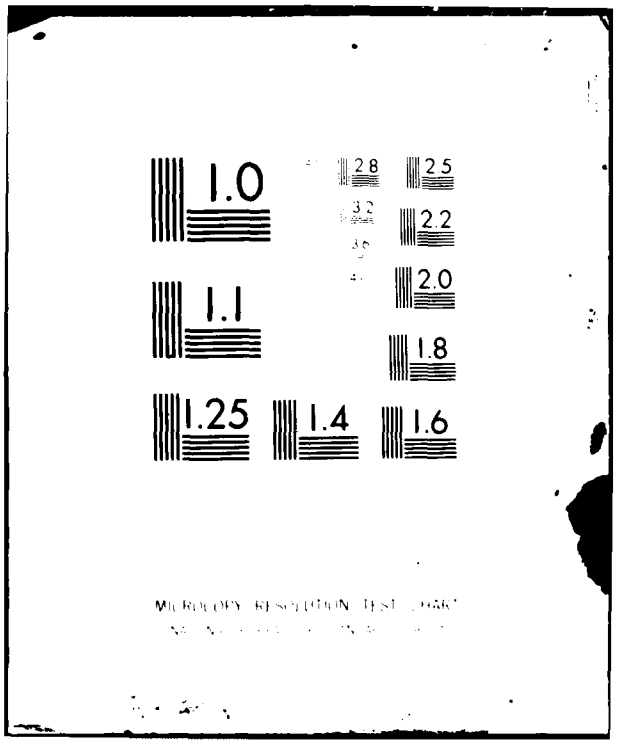
RDEV_TAB/	Read the contents of text object DEV_TAB into the current position
RTASK_TT/0..99	Read the text between lines 0 and 99 inclusive from text object TASK__IT into the current position. Neither line 0 nor line 99 need exist
RDEV_TAB2/@1100	Read the contents of text object DEV TAB2 into the position following line 1100 or the corresponding position if line 1100 does not exist.
R!1	Read the file or buffer named by the string contained in buffer 1 into the current position. If buffer 1 contained the seven character string "DEV_TAB", this command would be the same as the first example above.

Note: The text lines read into the current workbuffer are renumbered just as for a copy

AD-A109 981 COMPUTER SCIENCES CORP FALLS CHURCH VA F/G 9/2
ADA INTEGRATED ENVIRONMENT II COMPUTER PROGRAM DEVELOPMENT SPEC--ETC(U)
DEC 81 F30602-80-C-0292
UNCLASSIFIED RADC-TR-81-364-PT-2 NL



END
DATE FILMED
02-82
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST

A.9 W Write

Write (a portion of) the current work buffer to a text object.

If the text object is not empty, the output will be appended to the object.

WTPT_A/300..400	Write lines 300 thru 400 inclusive to TPT_A
WTQAA/5,100..200,13	Write lines 5, 100 thru 200 inclusive and line 13 to TQAA

A.10 B Buffer

Switch the current work buffer to a specified buffer.

BA	Switch to buffer A. The text in buffer A acts as the current work buffer
B<CR>	Switch back to the default work buffer.

A.11 L Locate Line

Position the cursor to a specified line

L100	Locate line 100, cursor set to line 100
L300	Locate line 300, cursor set to line 300

A.12 P Print Lines

Print specified lines

P200	Print line 200
P400..415	Print lines 400 thru 415

Volume 8

COMPUTER PROGRAM DEVELOPMENT SPECIFICATION

(TYPE B5)

COMPUTER PROGRAM CONFIGURATION ITEM

MAPSE Debugger

Prepared for

Rome Air Development Center

Griffiss Air Force Base, NY 13441

Contract No. F30602-80-C-0292

388

TABLE OF CONTENTS

SECTION 1 - SCOPE

1.1 IDENTIFICATION

This specification establishes the performance, design, development, and test requirements for the MAPSE debugging facility, the Debugger. The purpose of this specification is to define the Debugger being designed as part of the Ada Integrated Environment contract for RADC. This document will serve to communicate the functional design decisions that have been adopted, to provide a baseline for the detailed design and implementation phase and to identify any interfaces between the Debugger, the KAPSE system, and the remaining MAPSE tools.

1.2 FUNCTIONAL SUMMARY

The Debugger aids the Ada programmer in the checkout of Ada programs. Oriented toward interactive use, the Debugger allows the user to monitor an executing program by displaying the contents of the data and tracing the program flow. The user may interrupt an executing program, make program and data changes, set program breakpoints and collect path entrance and timing statistics. Statements, objects, types and expressions may be referenced using Ada symbology. Source lines numbers (or keys) may be used to relate the source program to the executing program. Although intended primarily for debugging at the Ada language level, the Debugger provides several directives that permit the user to perform operations at machine level, allowing instruction stepping, dumps (relocatable and absolute), patching, and examination of hardware registers.

The Debugger interfaces are described in the remaining sections. Briefly, those interfaces are:

1. Ada Debug Tables (ADTs) -- These tables are produced by the Complier as part of the relocatable object and are relocated and included into the load object by the Linker.

2. Load Objects -- the ADTs shall not be loaded into memory with a program but shall be accessed directly by the Debugger from the load object.

3. Ada Run-Time -- the Debugger must be cognizant of the conventions for subprogram calls, space management, exceptions, and tasking.

4. APSE Command Language Interpreter -- The Debugger shall form a part of the APSE Command Language Interpreter (ACLI) to permit the user to have full access to the APSE commands while debugging. The Debugger shall operate synchronously with the user`s program (process).

SECTION 2 - APPLICABLE DOCUMENTS

2.1 PROGRAM DEFINITION DOCUMENTS

1. Reference Manual for the Ada Programming Language, July 1980
2. Requirements for Ada Programming Support Environment, "STONEMAN", February, 1980.
3. Statement of Work, Contract No. F30602-80-C-0292, 80 Mar 26.

2.2 INTER-SUBSYSTEM SPECIFICATIONS

4. System Specificaton, 15 March 1981.
5. Volume 1, Computer Program Development Specification for CPCI KAPSE Framework.
6. Volume 2, Computer Program Development Specification for CPCI KAPSE Data Base System.
7. Volume 3, Computer Program Development Specification for CPCI APSE Command Language Interpreter.
8. Volume 4, Computer Program Development Specification for CPCI MAPSE Configuration Manager.
9. Volume 5, Computer Program Development Specification for CPCI MAPSE Compiler.
10. Volume 6, Computer Program Development Specification for CPCI MAPSE Linker.
11. Volume 7, Computer Program Development Specification for CPCI MAPSE Editor.

2.3 MILITARY SPECIFICATIONS AND STANDARDS

12. MIL-STD-483, Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 1 June 1971.
13. MIL-STD-490, Specification Practices, 30 October 1968.

2.4 MISCELLANEOUS DOCUMENTS

14. DIANA Reference Manual, 5 February 1981.
15. CSTS GPS Reference, Vol. 1, General.

16. DECsystem-10 Assembly Language Handbook, DDT.

17. MULTICS Programmers Manual-Commands and Active Functions.

SECTION 3 - REQUIREMENTS

3.1 INTRODUCTION

This section provides the general description, identifies the external and internal interfaces, provides the functional requirements and presents the internal characteristics of the Configuration Item identified as the Debugger.

3.1.1 General Description

The MAPSE Debugger operates as an independent subprogram of the APSE Command Language Interpreter (ACLI) and permits the user to observe and or change the operation of an executing program. The Debugger directives may be entered immediately after a program has been loaded but before its execution, or after execution has commenced by interrupting its operation. The Debugger shall operate with privileges that permit it to access and modify another process's instruction and data (after validation of the user's right to do so). The Debugger user interface shall support both immediate execution of directives or the implanting of directives (or primitive Ada statements) at selected points in the Ada program for execution when the implant location is encountered during the course of program execution.

The Debugger shall be reentrant and may operate on any of a user's active processes. The Ada Debug Tables (ADTs) shall be accessed by compilation unit (or active scope within a compilation unit) from the load object as needed to minimize memory requirements. The ADTs shall contain sufficient information about a program to allow the user to reference any program entity symbolicly in Ada syntax.

3.1.2 Peripheral Equipment Identification

The Debugger shall interface with peripheral equipment only as a potential source for its directives and to output the response to its directives. However, since this interface will be through the Ada Standard I/O Package, the devices shall be transparent to the Debugger.

The Debugger must be aware of the instruction set and hardware addressing architecture of the target computer of the programs being debugged. Initially the computer equipment of concern is identified as the IBM 370 and the Interdata 8/32.

3.1.3 Interface Identification

The Debugger interfaces are identified as the ACLI, the KAPSE Data Base System (KDBS), the KAPSE Framework (KFW), the Ada Compiler, the Linker, and the user through the Debugger Directive Language.

3.1.4 Function Identification

The Debugger is functionally grouped for specification purposes into four segments. These segments are identified below.

Initialization

This segment is used to establish the relationship of the Debugger and the process to be debugged (referred to hereafter as the DP). Part of this function is to initialize the description of DP and access the appropriate sections of the ADT.

Directive Recognition

The functions performed under this grouping are the directive parsing, establishing the ADT environment for name referencing, and evaluating directive expressions.

Directive Processing

The third grouping comprises the performance of the actions required by the individual Debugger directives. Among the functions that may be performed are display of program variables and hardware register contents, dumps of Ada program instructions or data in symbolic and machine format, recording of various execution statistics and Ada program data for subsequent analysis or data reduction, and searching of the Ada program for particular values or address references.

Program Execution

The final logical segment of the Debugger controls and monitors the execution of the subject program. Its functions are to perform various program tracing as selected by the user, to interface with the directive processor when implanted directives and breakpoints are encountered during program execution, to continuously monitor indicated variables or addresses for changed values (wild store traps), to collect path entrance and timing information, and perform any statement or instruction simulation necessary to perform controlled or stepped execution.

3.2 *FUNCTIONAL DESCRIPTION*

This section describes the functions of the Debugger, the program and equipment relationships and interfaces identified above, and the input/output utilization in the Debugger.

3.2.1 Equipment Description

The host computers upon whose definition the Debugger depends are the IBM VM/370 and the Interdata 8/32.

The directive interface uses the Ada character set. Some user mechanism must be available such as escape or break to effect a user generated process interruption. However, this particular character or mechanism is handled by the terminal handlers and is unknown to the Debugger.

3.2.2 Computer Input/Output Utilization

Directives are assumed to exist in the standard input file. Listing output is written to the standard output file and diagnostic messages are directed to the standard error file. The debug process's ADTs are read from the load object.

Strictly speaking, the Debugger has almost no interface to the computer upon which it is hosted. However, since the Debugger must, of course, understand inherently the target computer for which the program to be debugged has been compiled, and since the target is often the host computer, the distinction is academic. The target dependencies shall be isolated as follows. The representation of the target memory image on the host shall be described in

3497

a package. Implant definitions shall be reserved to the Implanter function. The remaining dependency is the format of target assembly language instructions in the dump processor.

3.2.3 Computer Interface Block Diagram

See Figure 3-1.

3.2.4 Program Interfaces

This section describes the Debugger interfaces and their purposes.

APSE Command Language Interpreter (ACLI)

The Debugger forms a part of this MAPSE tool and is invoked as a result of the ACLI encountering a Debugger directive. The ACLI and the Debugger cooperate during the processing of directives to permit each to utilize the services of the other. In particular, command macros, condition execution and transfer of control within commands is the responsibility of ACLI. References to, and evaluation of, Ada variables in ACLI commands are the responsibility of the Debugger.

KAPSE Data Base System (KDBS)

The KDBS is called directly and through the Standard I/O package to perform directive input from standard input and to output various Debugger responses to standard output and standard error.

KAPSE Framework (KFW)

The KFW is called to load the ADTs from the load object of the process to be debugged (referred to as DP); to bind the DP and the ACLI process together as necessary to ensure that both are available to the other during the debugging operations; to control the initiation, priority and resumption of DP; to establish addressing access, memory mapping, and memory protection for DP; to acquire additional workspace; and to query and manipulate Process Control Blocks (PCBs) in order to test, report on and change DP's execution context, i.e., its register settings, program counter, stack pointers, etc.

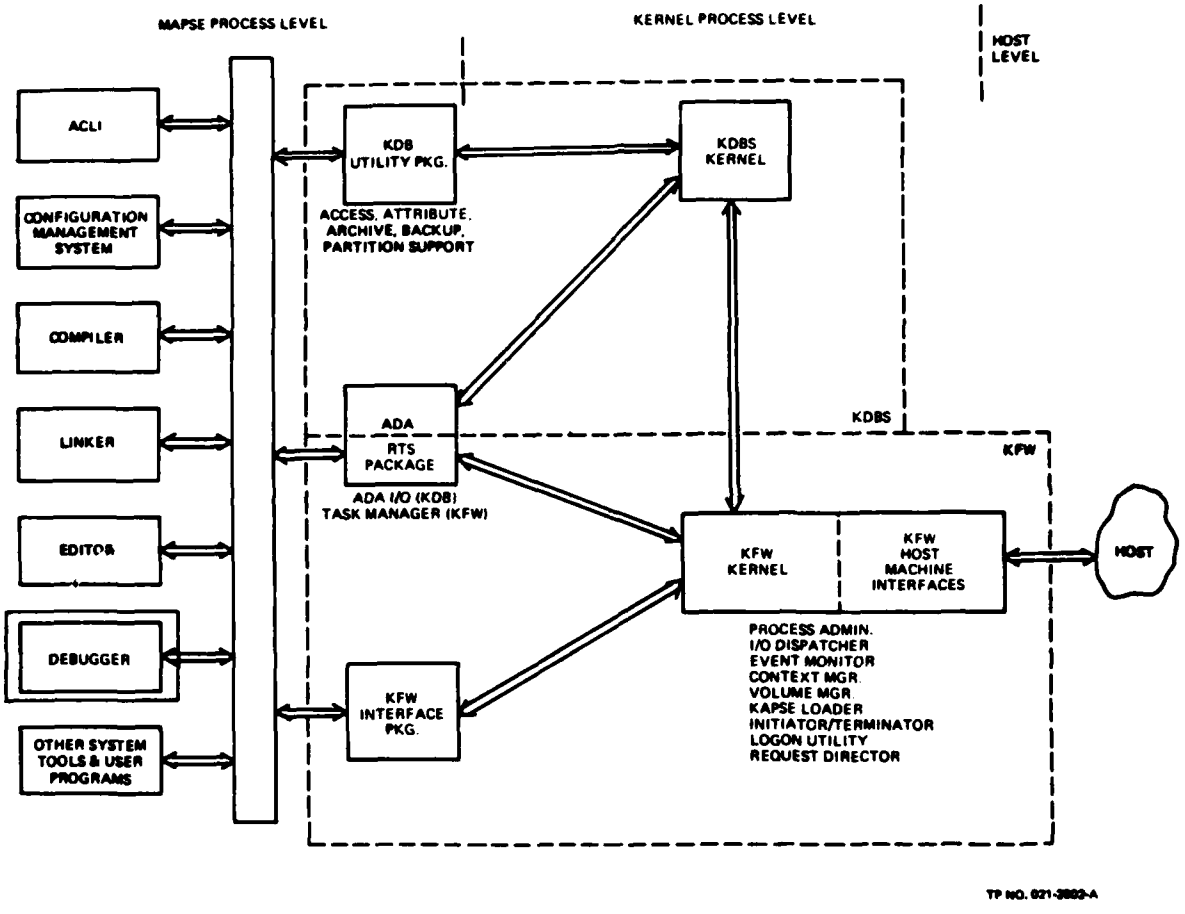


Figure 3-1. Interface Diagram

399

Ada Compiler

The Compiler produces the ADTs used by the Debugger to relate the DP code and data to the Ada source names and attributes. The code produced by the compiler represents another interface that the Debugger must understand to perform its functions. These interfaces embody the conventions followed by the executing program for subprogram calls and management, for Ada tasking, for local and heap data referencing, for register usage, and as a result of optimization.

Linker

The Linker relocates any addresses occurring in the ADTs and produces the load objects accessed by the Debugger.

Simulator

The Debugger is designed to interface with future target programs being executed via simulation. The implants inserted shall cause the simulator to relinquish control to the Debugger. All interfaces with the target program shall be in the target image as represented on the host. The existence of the simulator shall be transparent to the Debugger.

User

The Debugger functions are requested by utilizing a Debugger Directive Language (DDL). Ada program expressions, statements and objects are referenced in the DDL using Ada compatible syntax. The DDL is oriented toward usage in an interactive environment but is accepted identically from interactive terminals, directive objects or batch devices.

400

Program Interface Diagrams

The Debugger has two logical relationships that can be represented by separate diagrams. The first represents the interfaces resulting from communication between tools via the KDB objects. The second is the interface of the user to the debug process through the ACLI and the Debugger. Those diagrams are shown below.

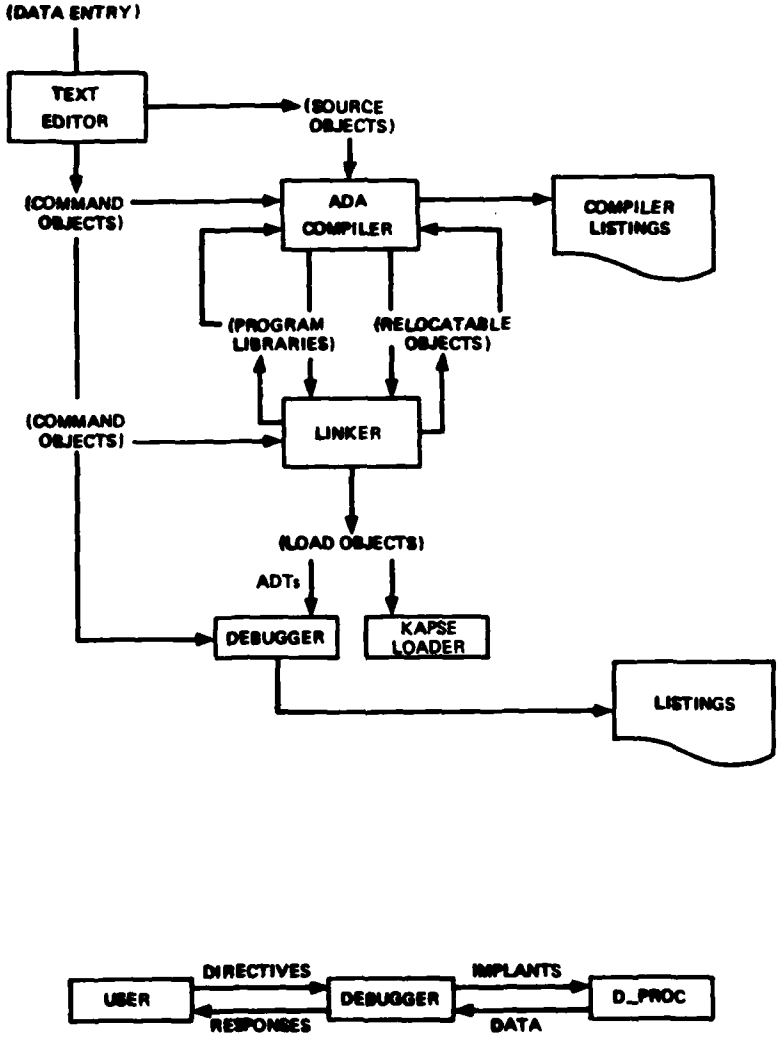


Figure 3-2. Debugger Interfaces

Debugger Directive Language

The Debugger utilizes a very simple directive language but supports a rich Ada expression capability.

The general format of the directives are:

[label:] action{,option ...} operand ,...

Each directive may have an optional label. A label is a name beginning with a letter and having zero or more additional alphanumeric or underscore characters. The action is a predefined Debugger word made up of one or more letters. All actions may be abbreviated to one or more of the initial characters of the action word. The operands are made up of values, names, expressions, or lists of the foregoing whose meaning depend upon the action indicated. The options are represented as a series of single letters amplifying the action to be performed. The directives are:

load obj_name [(parameters ,...)]

This directive is used to create a process and to ready the load object named for execution. Execution shall not begin until a goto is encountered at which time the parameters shall be passed as if a call of the object name had occurred as an ACLI command.

goto [address] [, { s|i } integer_expression]

This directive is used to start or resume execution of a program. If the operand is missing, the Debugger shall begin execution with the main program, if no prior execution had occurred, or at the current execution address if execution had been interrupted. The Debugger shall permit execution to begin at a user specified address but cannot insure that scope elaborations are valid or that the optimizer has not made some assumptions about the program that would be incompatible with the indicated transfer. Use of operands with this directive are not recommended when debugging at the Ada level. The user may request that execution continue for a specific number of statements or target instructions by using the s or i

options; the integer expression value is used as the number to execute. Declarations in an Ada program are counted as statements for the purpose of this directive.

help [action]

This directive is used to offer the user on-the-spot help with the Debugger directives. If the action is omitted, the Debugger shall display a summary of the directives available and their purpose. If an action is specified, its format and detailed usage shall be displayed.

insert { address ; " directive ;... " } | ?

This directive is used to cause the execution of a set of directives each time the specified location is executed. By implanting a stop directive, an execution breakpoint is effected and the Debugger shall solicit more directives. When an insert is made, the Debugger responds with a number that it associates with the insert. This number may be used to identify the insert for later disabling. If the question operand is present, the Debugger shall list the number and location of each active implant.

cancel integer ,...

Cancel the implant(s) identified.

stop[,s|p] { expression }

When inserted as an implant, this directive shall cause execution of the program to be suspended and Debugger directives to be requested. This directive has no effect when encountered other than as an implant. The value of the expression shall be printed when the stop is encountered. If the "s" option is selected, the execution context status shall be displayed. The p option requests a walkback printout of the active procedure call chain.

trace[,{p | f | s | a | i | w | o }] {address ,...} [,file]

This directive causes tracing printout as the program executes. The options are used to control the level of trace desired or to turn the trace off. The options are:

p Issue message when a procedure, function, package or block is entered or exited. On entrance, the in and inout parameters shall be printed; on exit, the inout and out parameters or function results shall be printed.

f Used to cause a flow trace. A message shall be printed when any transfer of control occurs (includes calls, gotos, ifs, cases, exceptions, etc.).

s Traces each statement executed and the values of any variables set as a result of their execution.

a Traces at the instruction level and lists the assembly instructions executed and any hardware registers changed as a result of the execution.

i Traces the results of I/O operations performed using the Ada Standard I/O Package. On read or write operations, the record contents will displayed.

w Prints the current walkback of active subprogram calls and the parameters passed.

o Turns the trace indicated in this directive off. If no trace indicator is indicated, all traces are turned off.

trap[,s] variable | address_range
trap[,o] [number]

This directive allows the user to monitor an address or variable for a changed value. When the directive is encountered, the contents of the variable, or address(es), are recorded. At every

reentrance to the Debugger, the contents will be examined. In addition, the contents will be checked at subprogram entrances and exits. In either case, if the contents have changed since the prior recorded value(s), the contents are displayed and the new values are recorded. If the s option is selected, DP will be suspended and the Debugger will be put in control; otherwise, execution will continue.

Each trap directive will be assigned an implant number which will be displayed. The o option form of the trap directive is used to remove an existing trap. If the identifying trap number is omitted, all traps are removed. Where the host computer supports store traps, or where DP is being executed under simulation control, the tests may be performed more frequently.

variable := expression | @

This is used to set a variable to a new value or to display its address. If the expression is present, the variable shall be assigned the value of the expression. If the '@' is present, the current address of the variable shall be displayed.

expression ?

This directive is used to display the value of a variable or an expression. The value shall be displayed using Ada symbology according to its type. If the expression is a variable, applicable indices and component names will be displayed.

dump[,{ i | r | b | o | h | c | a | s }] address_range

Dump is used to display memory in a selected format. If the format is omitted, the dump shall be printed in octal or hex, whichever is native to the target, plus the character representation. Addresses shall be displayed both relative to the compilation unit and its location counters and in absolute form. An option may be used to specify the dump format. The format options are:

i integer

r real

b boolean or binary depending upon variable size

o octal

h hexadecimal

c character

a assembly level instructions

s hardware status -- condition codes, registers, etc.

use { library_unit | package_name } ,...

This directive is used to specify the default qualification to be used for name resolution.

search address_range, { expression | range } [, mask]

This directive is used to search a range of memory for a particular value or for a value that lies within a specified range. Additionally, the searched words may be extracted using the indicated mask before applying the comparison.

{ flow | time } [,p|f|r] subprogram ,...

These two directives are used to count flow path entrances (flow) or to accumulate process time utilization (time) for the subprograms specified. If the p option is selected or no option is indicated, data will be collected on a subprogram basis; the f option collects on individual flow paths (see trace directive). The r option shall produce a report of the information collected for the subprogram(s) indicated.

record[,r|w] file , { variable ,... }

This directive is used to perform a simple record read or write of the variable using the file specified. The variable recorded shall be identified sufficiently in the file to permit an independent tool, using just the file and the ADTs from the associated load object, to data reduce the value. The r option is used to set a variable from a value in the file; the w option is used to write its value to the file. This facility is intended to support scaffolding and environment simulation.

print text_object [line_range]

This facility is used to print a specified line range from a text file such as an Ada source object. Since ACLI commands may be interspersed with Debugger directives, a user may invoke the Editor to make source modifications as debugging progresses.

quit

This directive causes the Debugger to return control to the ACLI. All traps, implants and connections to DP are disabled. The debugged process shall resume execution unless it has completed.

The expressions supported by the Debugger are expressed using Ada syntax as defined in the Ada reference manual. The operators and built-in functions supported shall be at least the following;

On records and arrays -- component selection, slicing and equality tests.

On scalar type variables and expressions --

```
+  -  *  /  mod  rem
<  >  =  /=  <=  =>
and  or  xor  not  in
&  **  abs  new  pred  succ
```

Aggregates shall be allowed. In addition, time operations and the val and image attributes shall be supported. The Ada abort, raise and delay statements are also permitted as directives.

3.2.5 Function Description

The following paragraphs are composed of two parts: the first is a paraphrasing of the explicit Debugger requirements as outlined in the above referenced Statement of Work (SOW); the second are those additional requirements that are believed necessary to provide a consistent, user-oriented and integrated development system for embedded computer applications.

3.2.5.1 SOW Requirements

These paragraphs describe the interpretation of the explicit SOW requirements.

The MAPSE system shall include facilities to assist batch and on-line users in discovering, finding and correcting Ada program errors. The Debugger must support debugging at the Ada level in that user input and Debugger output shall reference Ada source program statements by name or number (line number), Ada symbols, and variable names. Ada variable values shall be specified and referenced in Ada source format.

The Debugger must support all Ada language features including Ada tasking.

The Debugger must permit association of the load form of an Ada program with its corresponding source. Although this does not imply source level interpretation, facilities must be supported to access and list the related source, by accessing the original source object.

The Debugger must support the initialization and dynamic setting of conditional breakpoints and the single step execution on both an instruction and a statement basis.

The Debugger shall permit display and modification of Ada program variables and constants in machine and symbolic scalar type format at the user's option.

Tracing of program flow, including the display of subprogram argument values and names, shall be allowed. The user may also modify program flow; however, any change of flow from that represented by the program source is very dangerous because of optimizer assumptions, exception handling and scope elaboration requirements.

The user may select areas of memory for dumping.

3.2.5.2 Other Requirements/Capabilities

The following presents those additional functional capabilities which are believed necessary for a minimal Ada developmental system debugger.

No Special Compile Mode

Object programs shall, by default, be generated in a debuggable manner, i.e., the ADTs shall be produced automatically by the Compiler and passed on by the Linker. No hooks shall be generated in the code that would expand the size of a program to be debugged. The ADTs shall be associated with an executable program but shall not be part of the loaded data. The Debugger shall access the information from the load file. Similarly, the Debugger shall use the technique of instruction implantation and memory protection to control the debugging functions rather than rely upon compiler generated hooks and simulation.

Performance and Verification Support

The Debugger shall provide support for recording path entrance counts and timing statistics to facilitate program tuning and path entrance verification. The user may request program status reports; this shall include execution accounting information, path frequency counts, current program counter and special machine register values.

Directive Implantation

In addition to interactive directive execution, the Debugger shall permit the implantation of any Debugger directive for processing during program execution.

Dynamic Interruption

The normal mode of program checkout will be to load a program, perform some initial setup of breakpoints or implanted commands, and then to initiate interactive execution. An equivalently important requirement, however, is to permit interruption of an executing program that may be showing signs of aberrant behavior. A requirement for embedded debugging code can never be completely anticipated and seemingly thorough test cases seldom reveal problems of timing, space management, and language feature vs. optimization interaction.

Performance Characteristics

Performance considerations shall be a major concern of the Debugger design. Toward this end, the technique selected to facilitate program control is via instruction implantation rather than interpretation. Likewise, to prevent a space impact due to an increased memory space requirement to contain the ADTs, the ADTs shall be read in sections as needed.

Reliability

The MAPSE Debugger shall be designed to be reliable, to obey Ada strong typing and visibility rules, and to prevent violation of multiple user address spaces (as might occur for shared code).

3.3 DETAILED FUNCTIONAL REQUIREMENTS

This section describes in detail the functional processing to be performed by the Debugger. The Debugger is organized to be driven by user directives. The directives are processed by a common scanner (Director), any expressions are computed, and one of the eighteen directive processors is invoked. The processing of the directive is completed by the processor and control is returned to the Director.

A major partition of the Debugger is the Implantor. The Implantor is responsible for inserting all DP breakpoints, traps, or directive implants. A second major function of the Debugger is performed by the Executor. The Executor is responsible for any operation involving the activation of DP's code. When required the Implantor will ready the DP for execution and suspend the Debugger process until an implant is encountered by the DP durinmg the course of its execution. Figure 3-3 below depicts the structure of the Debugger.

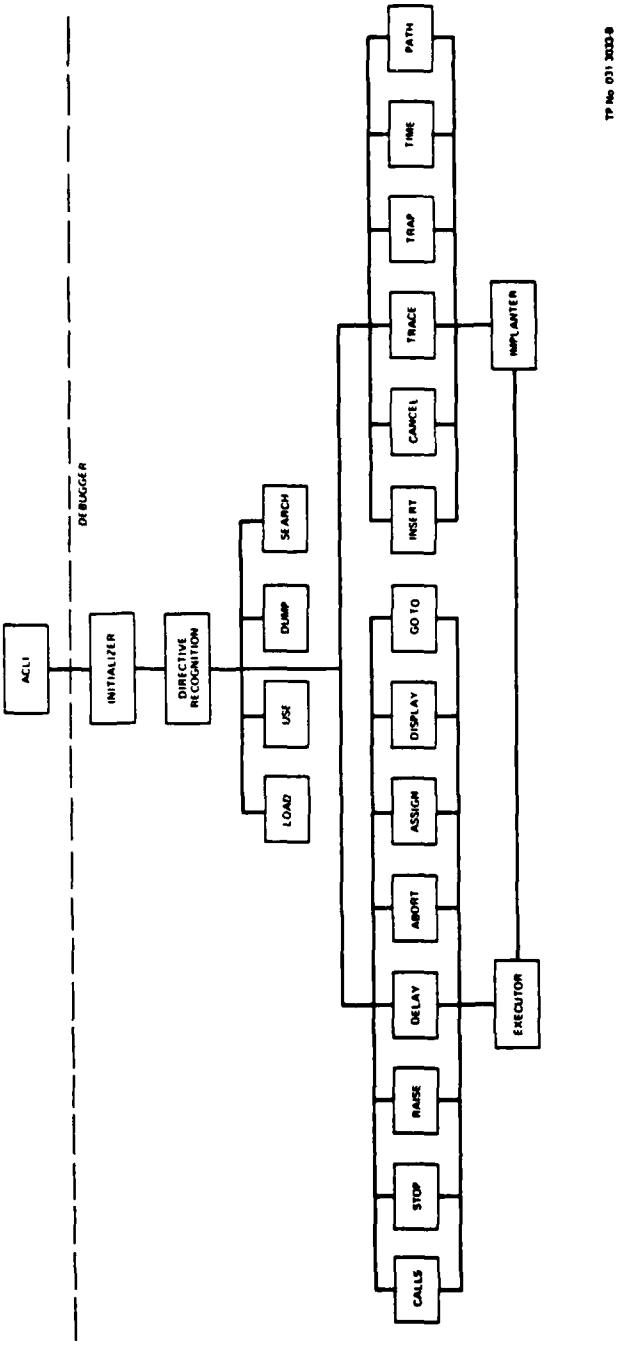


Figure 3-3. Debugger Structure

411

Before delving into the details of the Debugger components, a few definitions are necessary that will be referred to within the descriptions below.

Implants replace DP instructions and cause the Debugger to resume control when these implants are encountered during the execution of DP.

The context of DP comprises the collective description of its execution state. The context shall include the present program counter at the point of interruption, the stack status, the interruption cause, the task identity for multiple Ada task processes, and a list of active implants.

The Debugger is written entirely in Ada. The Debugger has no direct host computer interfaces; its intraprocess interfaces are serviced by the KFW. Small portions of the Debugger will be dependent upon characteristics of the target computer.

The Debugger is invoked as a procedure as follows:

```
DEBUG ( PROCESS-ID ) ;
```

where PROCESS-ID is the identity of the debug process (DP)

3.3.1 Initialization

The initial function of the Debugger is to initialize its data space, establish a co-process relationship with DP and include the DP address space(s) within its address space.

3.3.1.1 Inputs

DP - The process to be debugged.

3.3.1.2 Processing

The KFW is called to bind the Debugger (ACLI) process to DP to insure that the two processes are always swapped together. The DP is incorporated into the address space of the Debugger again through a KFW interface. Heap space of DP is acquired for working space and to contain the ADT sections temporarily in use by the Debugger. Directive processing is initiated.

3.3.1.3 Outputs

None.

3.3.1.4 Special Requirements

This area is sensitive to the host machine and underlying system architecture as the facilities for creating co-processes and address sharing may differ substantially.

3.3.2 Directive Recognition (Director)

This logical segment of the Debugger is responsible for the syntax analysis of the Debugger directives and driving the remaining Debugger functions.

3.3.2.1 Inputs

Standard input -- the input object or terminal from which the directives shall be read.

3.3.2.2 Processing

Each directive is lexically analyzed. If a label is encountered, the label is associated with the current directive object position. If the next token is not an action primitive, the name is assumed to be an Ada name or expression that must be resolved using the ADTs.

This name (or any other name occurring within the directive) is resolved in the following manner. If the name is fully qualified by a library unit, it shall be resolved from the ADTs. An attempt shall be made to resolve the name by using the visibility rules applied at compilation for the current active scope of the program or, alternatively, by resolving to a name made directly visible via the "use" directive.

Next any expressions encountered within the directive shall be parsed and their values computed. Control shall then transfer to one of the functions below according to the action requested.

3.3.2.3 Outputs

Any error encountered in the processing of the directive shall be reported on the standard error file.

414

3.3.3 Help

This paragraph describes the Debugger's directive assistance facility.

3.3.3.1 Inputs

An optional action word may be specified.

3.3.3.2 Processing

If no action is specified, the list of allowable actions and a brief summary of their purposes is produced. If a specific action is identified, a detailed explanation of the directive's syntax and usage is displayed.

3.3.3.3 Outputs

The stated output is written to standard output.

3.3.4 Use

This section describes the directive used to establish the default scope qualification for Ada program name resolution.

3.3.4.1 Inputs

The library unit or package qualification to be used as the default for name resolution. This name must be of a compilation unit.

3.3.4.2 Processing

This directive serves two purposes. The first is an external function used to allow simpler reference to names contained in the compilation unit or package most commonly referenced by the user during debugging. The second function is the mechanism by which this resolution is accomplished; i.e., the ADT for the indicated compilation unit is promoted to in-core residence both to support the resolution and to improve performance by permitting faster resolution of the high frequency references.

3.3.4.3 Outputs

None.

3.3.5 Goto

This paragraph describes the processing of the goto directive.

3.3.5.1 Inputs

An optional goto address may be supplied.

3.3.5.2 Processing

If an address is supplied, the context of DP is changed to reflect a revised program counter using the address specified. Execution of DP is then resumed by transferring control to the Executor.

3.3.5.3 Outputs

If an address is supplied which is out of the current context scope but in an active scope, a warning diagnostic is issued on standard error. If the address is in a currently inactive scope, a serious diagnostic is issued on standard error and control is returned to the directive processor.

3.3.6 Insert

The insert function is to perform the functions of the insert directive.

3.3.6.1 Inputs

Insert has two inputs: the first is the address at which the inserted directive string is to be placed, and the second represents the directives to be inserted.

3.3.6.2 Processing

The insertion address is located and the instruction(s) at the address are examined to insure safe implanting. If the instruction context will not allow insertion, e.g., a skip instruction, an address adjustment is requested from the user. These instructions are then replaced by an instruction which shall cause interruption of DP's execution and the Debugger to resume control. The replaced instruction(s) are saved with the implant directives for subsequent execution when the insert address is encountered during the course of DP execution.

3.3.6.3 Outputs

If no implant string is input, a list of the currently active implants is written to standard output. If an implant instruction conflict occurs at the address indicated an address revision is requested. When a successful implant can be made, an implant identity is indicated on standard output.

3.3.6.4 Special Requirements

The implanting is target dependent. Both the safe context for implanting and the implanted instruction required to place the Debugger back in control when the implant is encountered will change with each target.

3.3.7 Stop

The stop directive processing is described below.

3.3.7.1 Inputs

A stop expression may be optionally specified.

3.3.7.2 Processing

When this directive is encountered, the context program counter is noted both symbolically and by machine address on standard output. The Stop expression value, if present, is also displayed on standard output. If the 's' option is present, the context status is displayed. Control returns to the directive processor.

3.3.7.3 Outputs

The stop address and expression value are written to standard output as stated.

3.3.8 Display

This function is called as a result of "expression ?" appearing as a directive.

3.3.8.1 Inputs

A pointer to the ADT entry for the variable to be displayed and any associated indices or access variables are input.

3.3.8.2 Processing

If the variable is a scalar quantity, its name, any indices and the current value of the variables shall be displayed in the Ada symbolic representation for its declared type. If the variable is of a composite type such as array or record, each component shall be so displayed. Control shall then resume with the Director.

3.3.8.3 Outputs

Variable names and values shall be output to standard output.

3.3.9 Assignment

These paragraphs describe the processing of the assignment directive.

3.3.9.1 Inputs

The name and indices of the variables are input as well as the new value which the variable is to be assigned.

3.3.9.2 Processing

The value to be assigned is checked for constraint violations according to the type of the subject variable. If in violation, a diagnostic is issued and the assignment is not made. Otherwise the value is assigned to the variable and control returns to the Director.

3.3.9.3 Outputs

If a value or indices constraint violation is discovered, or if the variable's declaration has not been elaborated, or the variable has been delaborated, a diagnostic shall issued on standard error accordingly.

3.3.10 Trace

The processing of the trace directive is described in the following paragraphs.

3.3.10.1 Inputs

The trace option requested is input.

3.3.10.2 Processing

This directive causes internal trace options in the Debugger to be activated according to the trace option selected. If a trace limit address is specified, a trace-off directive is implanted at the address indicated. Processing returns to the Director.

3.3.10.3 Outputs

None.

3.3.11 Dump

The processing of the dump directive is described below.

3.3.11.1 Inputs

The address range to be dumped and the dump format to be used is input.

3.3.11.2 Processing

The address range specified is dumped in the format specified. The dump addresses shall be listed by library unit and location counter as well as by absolute value. If the format selected is assembly, each recognizable instruction shall be disassembled and operands shall be represented as relative to the compilation unit in which they occur.

3.3.11.3 Outputs

The symbolic dump output shall be listed on standard output.

3.3.12 Block Processing

These paragraphs describe the processing associated with both the timing and path entrance directives.

3.3.12.1 Inputs

The scopes to be monitored for timing or path entrance statistics collection are specified.

3.3.12.2 Processing

The processing of these two functions are very similar. Each basic block described in the ADT which is within the scopes indicated shall be implanted successively as execution proceeds. This is controlled by setting internal Debugger flags indicating that one or both of these basic block functions are required. If timing is to be collected or path entrances are to be counted, an implant is made at the beginning of the indicated scope. Implants are also inserted at block exits for timing accumulation. The actual timing functions are made by the Executor.

3.3.12.3 Outputs

None.

3.3.12.4 Special Requirements

The host and any underlying operating system must provide facilities for a relatively fine level of cpu time utilization for the timing statistics to be accurate.

3.3.13 Search

This function is to support the search directive.

3.3.13.1 Inputs

Three values are required for the search directive processing: the search address range, the search value or value range, and the mask to be applied.

3.3.13.2 Processing

This directive is primarily used when debugging at the machine level but is nonetheless quite useful. The function is performed by searching all words (or sets of words according to the search value type) for the value specified. If a value range is specified, the words are examined to determine if their value is within the range specified. If a mask is supplied, the words are extracted before the comparison.

3.3.13.3 Outputs

Every occurrence of a successful find is indicated on standard output by printing the address of success and, for a value range search, the value found.

3.3.14 Trap

The trap directive functions are described below.

3.3.14.1 Inputs

The address or variable to be trapped is input.

3.3.14.2 Processing

The indicated address or variable is recorded as a trap and the current value in the address or variable is saved. Additionally, an address limit breakpoint is inserted at the end of the scope of the variable (for non-static variables only). A trap implant is made into the scope prologue and epilogue routines to test for changed values in any trapped addresses and variables. When a trap has a changed value, the execution state, the trap, and the new value are listed and execution continues. When the trap limit address or scope exit is encountered, the trap is disabled.

3.3.14.3 Outputs

When a trap directive is encountered the current value is printed on standard output. When a trap has changed values, the trap and its new value are listed.

3.3.14.4 Special Requirements

Some computers will support such store traps directly providing a finer degree of trapping.

3.3.15 Load

The load function is responsible for creating a user process in response to a load directive request.

3.3.15.1 Inputs

The inputs to the load function are:

load object -- program to be executed

Parameters -- the parameters to be passed to the process

3.3.15.2 Processing

Load shall call the KFW to create a process. This process shall be assigned a priority less than that of the Debugger to prevent its execution. The parameters shall be entered in the user's space and directive processing shall continue.

3.3.15.3 Outputs

Load shall generate a diagnostic of standard error if the load object does not exist or the user does not have execute access to the object.

3.3.16 Call

This functional unit is responsible for the call directive.

3.3.16.1 Inputs

The call directive inputs are:

Procedure name -- procedure to be called

Parameter list -- of parameter to pass to the procedure

3.3.16.2 Processing

Call sets up a call to the subject procedure in the DP's address space. The parameter conventions for the target shall be obeyed. The Implantor is called to cause a return to the Debugger when the procedure completes its execution. The Executor is invoked to begin the execution of DP at the procedure call established.

3.3.16.3 Outputs

If the procedure has any output parameters, the values shall be assigned as necessary after returning from the procedure.

3.3.16.4 Special Requirements

Since the call to be setup is target-dependent, this module will require modification for retargeting.

3.3.17 Cancel

This module is called to process a cancel directive.

3.3.17.1 Inputs

The list of implant numbers to be cancelled in input to cancel.

3.3.17.2 Processing

After validating that the implant number is valid, Cancel simply calls the Implantor with the list of implants to be cancelled.

3.3.17.3 Outputs

If the implant is invalid, a diagnostic shall be written to standard error.

3.3.18 Record

This unit is called to perform the record directive functions.

3.3.18.1 Inputs

File name -- file to be read or written

Options -- to indicat read or write

Variable list -- list of variables to be transferred

3.3.18.2 Processing

Record attempts to open the file requested if it is not already opened. If the read option is selected, the value is read in and assigned to the variables in the order of their appearance in the list. If the write option is requested, the load object name, the qualified variable name, any indices associated with the variable, and the variable value(s) are written to the file.

3.3.18.3 Outputs

File name -- The value of the variable is written to the file if the write option is requested.

3.3.19 Executor

This the most complex of the functions to be performed by the Debugger and comprises the bulk of the Debugger processing. This function is the control of the execution of DP.

3.3.19.1 Inputs

There are not specific inputs to the Executor but many global parameters and implants are used to control its performance. Amongst these flags are the tracing parameters, the execution granularity, e.g., single step, timing and path entrance options.

3.3.19.2 Processing

In the absence of any ongoing execution control, execution shall simply begin by readying DP for scheduling and the Debugger process shall be put to sleep. However, in the presence of any of the tracing, stepping, or statistics collection functions, the Executor must insert an appropriate implant into the code prior to resuming the execution of DP. When the next implant or breakpoint is encountered, the Executor shall resume control and react according to the implant encountered. The different causes and resulting actions are described below:

Directive Implant

The execution state is saved and the Director is called recursively. When completed, execution resumes.

Stepping Breakpoint

If tracing is active for the stepping level breakpoint encountered, an appropriate flow trace message shall be printed. If the stepping range has been exceeded, the stepping flags are turned off and control returns to the Director. Otherwise the next step address breakpoint is implanted and DP execution resumes.

Timing breakpoint

If a scope entrance has been encountered, the current process clock value is saved and then reset to zero. When a scope exit is encountered, the clock value is added to the previous accumulated timing for the scope and the clock is reset to the value saved at scope entrance. In either case, the next breakpoint is set and execution is continued.

Path Entrance

When a basic block entrance is encountered, the path entry count is incremented, the next breakpoint is set and execution resumes.

Address Limit

Various directives have address limits over which they apply. These limits are implemented by inserting an address limit breakpoint. When such a breakpoint is encountered, the affected action is disabled, and execution resumes.

For all breakpoints, a check is made to determine if a tracing event has occurred and the appropriate output is produced. Likewise, if a trapped value has changed, generate trap output accordingly.

3.3.19.3 Outputs

None.

3.3.20 Implantor

This module performs all implants into the target process.

3.3.20.1 Inputs

Implant type -- type of breakpoint required

Address -- location of the implant

3.3.20.2 Processing

The Implantor shall be called by all directive processors and by the executor for the insertion or cancelation of any implant. This function is target-dependent because the instruction context of an implant address may not permit the implant. The Compiler provides an address with each statement for safe implanting. This address is used in the user has not specified a particular address. The Compiler supplied address may be used as an anchor point to locate legal implant location in the region if the instructions are variable length.

3.3.20.3 Outputs

None.

3.3.20.4 Special Requirements

All target dependencies associated with implants are isolated in this module to ease the task of retargeting.

3.4 ADAPTATION

This section describes any dynamic adaptation that might be required to parameterize the Debugger's operational system environment, system parameters, and capacities.

3.4.1 General Environment

The Debugger relies heavily on the functionality of inserted breakpoints. This functionality can be machine-dependent and the Debugger will have to adapt to any changes to the canonical functionality described in Paragraph 3.3.14. The Debugger, as well as the rest of the ACLI, also must respond to user-generated "breaks" or interrupts, whose characteristics can also be machine-dependent.

3.4.2 System Parameters

System parameters do not directly affect the size of the process being debugged, since the Debugger, within the ACLI, executes as a separate process. However, on systems with very limited memory, if the Debugger and the debugged process cannot both be in main memory at the same time, the performance of the Debugger will be severely constrained.

3.4.3 System Capacities

The Debugger will require space to retain timing and path counters for each block or subprogram requested. There will be parameterized limits on the number of counters allowed.

3.5 CAPACITY

Not applicable.

SECTION 4 - QUALITY ASSURANCE PROVISIONS

4.1 INTRODUCTION

This section contains the requirements for verification of the performance of the Debugger. The test levels, verification methods, and test requirements for the detailed functional requirements in Section 3 are specified in this section. The verification requirements specified herein shall be the basis for the preparation and validation of detailed test plans and procedures for the Debugger. Testing shall be performed at the subprogram, program (CPCI), system integration, and acceptance test levels. The performance of all tests, and the generation of all reports describing test results, shall be in accordance with the Government approved CPDP and the Computer Program Test Procedures.

The verification methods that shall be used in subprogram and program testing include the methods described below:

1. Inspection - Inspection is the verification method requiring visual examination of printed materials such as source code listings, normal program printouts, and special printouts not requiring modification of the CPCI. This might include inspection of program listings to verify proper program logic flow.

2. Analysis - Analysis is the verification of a performance or design requirement by examination of the constituent elements of a CPCI. For example, a parsing algorithm might be verified by analysis.

3. Demonstration - Performance or design requirements may be verified by visual observation of the system while the CPCI is executing. This includes direct observance of all display, keyboard, and other peripheral devices required for the CPCI.

4. Review of Test Data - Performance or design requirements may be verified by examining the data output when selected input data are processed. For example, a review of hard copy test data might be used to verify that the values of specific parameters are correctly computed.

435

5. Special Tests - Special tests are verification methods other than those defined above and may include testing one functional capability of the CPCI by observing the correct operation of other capabilities.

These verification methods shall be used at various levels of the testing process. The levels of testing to be performed are described in the paragraphs below. Data obtained from previous testing will be acceptable in lieu of testing at any level when certified by CSC/SEA and found adequate by the RADC representative. Any test performed by CSC/SEA may be observed by RADC representatives whenever deemed necessary by RADC.

Table 4-1 specifies the verification method for each functional requirement given in Section 3 of this specification. The listing in Table 4-1 of a Section 3 paragraph defining a functional requirement implies the listing of any and all subparagraphs. The verification methods required for the subparagraphs are included in the verification methods specified for the functional requirement. Acceptance test requirements are discussed in Paragraph 4.3.

4.1.1 Subprogram Testing

Following unit testing, individual modules of the Debugger shall be integrated into the evolving CPCI and tested to determine whether software interfaces are operating as specified. This integration testing shall be performed by the development staff in coordination with the test group. The development staff shall ensure that the system is integrated in accordance with the design, and the test personnel shall be responsible for the creation and conduct of integration tests.

4.1.2. Program (CPCI) Testing

This test is a validation of the entire CPCI against the requirements as specified in this specification.

Table 4-1. Test Requirements Matrix

SECTION	TITLE	INSP.	SPEC.	DEMO.	DATA.	SECTION NO.
3.3.1	Initialization		X			4.2.2
3.3.2	Directive Recognition		X			4.2.2
3.3.3	Help				X	4.2.1
3.3.4	Use				X	4.2.1
3.3.5	Goto				X	4.2.1
3.3.6	Insert				X	4.2.1
3.3.7	Stop				X	4.2.1
3.3.8	Display				X	4.2
3.3.9	Assignment				X	4.2.1
3.3.10	Trace				X	4.2.1
3.3.11	Dump				X	4.2.1
3.3.12	Block Processing		X		X	4.2.1
3.3.13	Search				X	4.2.1
3.3.14	Trap				X	4.2.1
3.3.15	Load				X	4.2.1
3.3.16	Call				X	4.2.1
3.3.17	Cancel				X	4.2.1
3.3.18	Record				X	4.2.1
3.3.19	Executor		X			4.2.2
3.3.20	Implantor		X			4.2.2

437

CPCI testing shall be performed on all development software of the Debugger. This specification presents the performance criteria which the developed CPCI must satisfy. The correct performance of the Debugger will be verified by testing its major functions. Successful completion of the program testing that the majority of programming errors have been eliminated and that the program is ready for system integration. The method of verification to be used in CPCI testing shall be review of test data. CPCI testing shall be performed by the independent test team.

4.1.3 System Integration Testing

System integration testing involves verification of the integration of the Debugger with other computer programs and with equipment. The integration tests shall also verify the correctness of man/machine interfaces, and demonstrate functional completeness and satisfaction of performance requirements.

System integration testing shall begin in accordance with the incremental development procedures as stated in the CPDP. Final system integration shall occur subsequent to the completion of all the CPCIs comprising the MAPSE system. Two major system integration tests shall be performed: one for the IBM VM/370 implementation and one for the Interdata 8/32 implementation. The method of verification used for system integration testing shall be the review of test data.

The test team shall be responsible for planning, performing, analyzing monitoring, and reporting the system integration testing.

4.2 TEST REQUIREMENTS

Quality assurance tests shall be conducted to verify that the Debugger performs as required by Section 3 of this specification. Table 4-1 specifies the methods that shall be used to verify each requirement. The last column refers to a brief description of the specified types of verification as given below. Test plans and procedures shall be prepared to provide details regarding the methods and processes to be used to verify that the developed CPCI performs as required by this specification. These test plans and procedures shall contain test formulas, algorithms, techniques, and acceptable tolerance limits, as applicable.

4.2.1 Review of Test Data

Scripts are prepared to exercise each command and the results are compared to the test data.

4.2.2 Special Tests

Each function is tested as a part of the debugging process.

4.3. ACCEPTANCE TESTING

Acceptance testing shall involve comprehensive testing at the CPCI level and at the system level. The CPCI acceptance tests shall be defined to verify that the Debugger satisfies its performance and design requirements as specified in this specification. System acceptance testing shall test that the MAPSE satisfies its functional requirements as stated in the System Specification. Acceptance testing shall be performed by review of test data.

These tests shall be conducted by the CSC/SEA team and formally witnessed by the government. Satisfactory performance of both CPCI and system acceptance tests shall result in the final delivery and acceptance of the MAPSE system.

SECTION 5 - DOCUMENTATION

5.1 GENERAL

The documents that will be produced during the implementation phase in association with the Ada compiler development are:

1. Computer Program Development Specification
2. Computer Program Product Specification
3. Computer Program Listings
4. Maintenance Manual
5. Users Manual
6. Retargetability/Rehostability Manual
7. MAPSE Tools Reference Handbook

5.1.1 Computer Program Development Specification

The final MAPSE Debugger B5 Specification shall be prepared in accordance with DI-E-30139 and submitted 30 days after the start of Phase II. A single document shall be prepared for the Debugger that defines the functional capabilities and interfaces. Any dependencies on the host and target shall be addressed in the document. Additionally, characteristics of potential hosts and targets which have had impact on the B5 specification shall be presented.

5.1.2 Computer Program Product Specification

A type C5 specification shall be prepared during the course of Phase II in accordance with DI-E-30140. This document shall be used to specify the Debugger design and development approach for implementing the B5 specification. This document shall provide the detailed description which

140

shall be used as the baseline for any Engineering Change Proposals. A single C5 shall be produced for the Debugger with different sections addressing the dependencies of the two host computers.

5.1.3 Computer Program Listings

Listings shall be delivered which are the result of the final compilation of the accepted Debugger. Each compilation unit listing shall contain the corresponding source, cross-reference and compilation summary. The source listing shall contain the source lines from any INCLUDEd source objects.

5.1.4 Maintenance Manual

A Debugger Maintenance Manual will be prepared in accordance with DI-M-30422 to supplement the C5 and compilation listings sufficiently to permit the Debugger to be easily maintained by other than the developer. The documentation shall be structured to relate quickly to program source. The procedures required for debugging and correcting the Debugger shall be described and illustrated. Sample run streams for compiling Debugger components, for relinking the Debugger in parts or as a whole, and for installing new releases shall be supplied. The data base shall be fully documented with pictures of record layouts where appropriate and the data algorithms shall be explained.

The Maintenance Manual shall be organized with a standard outline and separate parallel volumes shall be delivered which address the tailoring of the Debugger to a particular target or host computer. Debugging aids which have been incorporated as an integral part of the Debugger shall be described and their use fully illustrated. Special attention shall be given to the description of the maintenance mode operation of the Debugger used to aid in the pinpointing of Debugger problems.

5.1.5 Users Manual

A Users Manual shall be prepared in accordance with DI-M-30421 and shall contain all information necessary for the operation of the Debugger. Because of the virtual user interface presented by the ACLI and the Debugger directives, a single manual is sufficient for all host computers. Sample Debugger listings shall be included in the manual.

A complete list of all Debugger diagnostic messages shall be included with supplemental information chosen to assist the programmer in locating and correcting source program errors.

5.1.6 Retargetability/Rehostability Manual

In accordance with R&D-137-RADC and R&D-138-RADC, a manual shall be prepared which describes step by step the procedures for retargeting the Debugger to a different computer and transporting the Debugger onto a different host computer. Tips shall be provided which shall guide the developer module by module as to what may be used entirely or in part.

5.1.7 MAPSE TOOLS Reference Handbook

A MAPSE Tools Reference Handbook will be prepared which will provide a handy reference to the various MAPSE Tools. The Debugger directives will be summarized in this handbook.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.


DATE
FILMED
2-8